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.

Warehouses & Lakehouses (Snowflake, BigQuery, Redshift, Databricks)

Focus: Modern analytical storage engines. Comparing cloud data warehouses (Snowflake, BigQuery, Redshift, Synapse) and lakehouse technologies (Databricks with Delta Lake, etc.). Key concepts: separation of storage/compute, how Snowflake micropartitions work, BigQuery architecture (Dremel), how Databricks Delta provides ACID on data lake, etc. Possibly cost and performance tuning in each.

Why It Matters: Choosing the right analytical DB and using it effectively is core to DE. Also, lakehouse (like Delta Lake on S3) vs traditional warehouse differences are hot topics. Interviewers gauge if you understand new patterns vs old.

---

45. **(Easy)** What is **Snowflake** and why has it become popular for data warehousing?

Answer:

- **Snowflake** is a cloud-native data warehousing platform known for separating storage and compute and a fully managed service that runs on AWS, Azure, GCP. It uses an MPP architecture under the hood but with unique features like auto-scaling warehouses and cross-cloud capabilities.
- **Key features/why popular:**
  - **Separation of compute and storage:** You can scale up or down compute (called "virtual warehouses") independently of your storage sizedatacamp.comdatacamp.com. Storage is on cheap cloud storage (and Snowflake compresses and organizes it in micro-partitions).
  - **Auto-suspend and auto-resume:** Compute warehouses can shut off when idle (saving cost) and resume on query, enabling cost-efficient usage.
  - **Time Travel & Zero-Copy Cloning:** It keeps some history of data (time travel up to 90 days depending on config) allowing you to query as of past times or clone tables instantly without copying datadatacamp.comdatacamp.com.
  - **Fully managed:** No index management, vacuuming, or distribution keys to think about. It handles optimization internally. Very little DBA work needed.
  - **Performance:** Uses columnar storage, pruning (only reading micro-partitions relevant to query by min/max metadata)datacamp.com, results caching (so repeated queries return quickly from cache), and vectorized processing. It's fast for many analytic workloads.
  - **Concurrency:** You can spin up multiple warehouses for different users/workloads that don't contend (one team's queries won't slow another, they use separate compute).
  - **SQL compatibility:** Supports standard SQL, making it easy for teams to adopt.
  - It's become popular because it essentially offers a near-zero maintenance, scalable solution – companies don't have to manage infrastructure or tune as heavily, and it handles many workloads easily (BI dashboards, ad-hoc analysis, etc.).
- In summary, Snowflake's elasticity, features like time travel, and simplicity (plus marketing and being early to multi-cloud) made it a top choice for cloud DW.

What interviewers look for: Recognize Snowflake as a cloud data warehouse, articulate a few reasons like separation of compute/storage and ease of use. If mention micro-partitions or features, shows deeper knowledge.

Common pitfalls:

- Calling Snowflake a “database on S3” oversimplified (kind of true but it's a whole service).
- Not knowing at all (if a DE hasn't heard of Snowflake, that's a gap given its popularity).
- Overstating: “Snowflake is infinitely scalable” – it has great scale but not magical; but practically for most, it scales well.

Follow-up: How does Snowflake’s pricing model work? (Answer: charges for compute by the second (warehouse credits, scaled by size of warehouse cluster), and for storage by amount stored per month. Also data egress charges possibly. So separate storage and compute billing.)

---

46. **(Easy)** What is a **Lakehouse** architecture (e.g., Databricks Lakehouse) and how is it different from a traditional data warehouse?

Answer:

- A **Lakehouse** is an architecture that combines elements of data lakes and data warehouses to provide ACID transactions and schema enforcement on top of low-cost data lake storage. In simpler terms, it’s a data lake with warehouse-like management and performance features.
- Example: Databricks’ Lakehouse uses **Delta Lake** format on object storage (like S3/ADLS/GCS) which adds a transaction log enabling ACID operations, time travel, and concurrent writes on data lake filesdatabricks.com. So you can treat files in S3 similar to a database table (updates, deletes possible with consistency).
- Differences from traditional DW:
  - **Storage**: Lakehouse keeps data in files on distributed storage (parquet, delta files on S3, etc.), whereas a warehouse usually has its own internal storage format on its managed disks.
  - **Schema and Data Governance**: Historically data lakes were schema-on-read and loose, but lakehouse introduces schema enforcement and governance on top (like Delta can enforce schema, Hive metastore or Unity Catalog keeps schema info).
  - **Cost and Scalability**: Lakehouses leverage cheap object storage and can separate compute engines (Spark, SQL engines) that read the same data. A warehouse is often a monolithic system (like Redshift storage tightly-coupled to its compute nodes).
  - **Flexibility**: Lakehouse supports unstructured/semi-structured data more natively (since underlying storage is like a lake). One can run machine learning or SQL on same source. Traditional warehouses are SQL-centric for structured data.
  - **Examples**: Databricks Lakehouse, Delta Lake; also Apache Iceberg or Hudi provide similar lakehouse capabilities. Snowflake is now sometimes bridging into lakehouse by allowing external tables on lake data, etc.
- In a lakehouse, one engine (like Databricks or Spark) can do both large-scale ETL and BI queries on the same data. It tries to eliminate having separate copies for data lake and

data warehouse.
- Summarize: Lakehouse merges the reliability/performance of warehouses (via ACID, indexing, etc.) with the openness/scale of data lakes (files on cheap storage, accessible by multiple tools).

What interviewers look for: Understanding of concept – at least mention ACID on data lake or Delta Lake. And difference that data isn't locked inside a warehouse appliance.

Common pitfalls:

- Saying lakehouse is just "storing warehouse data on a lake" without mention of the tech (like transaction log).
- Not differentiating from data lake: must emphasize the added structure and features.
- If never heard term, could confuse with just "having both a lake and a warehouse." But in context they likely mean modern lakehouse pattern.

Follow-up: Name one open-source project enabling lakehouse capabilities. (Answer: Delta Lake (open-sourced by Databricks), Apache Iceberg, Apache Hudi. They all provide ACID table layers on data lakes.)

---

47. **(Medium)** What are **micro-partitions** in Snowflake?

Answer:

- **Micro-partitions** are Snowflake's internal data storage units. When Snowflake ingests data into a table, it organizes it into small contiguous units of storage (around 50 MB compressed each) sorted by order of insertionblogs.rajinformatica.com.
- Each micro-partition stores a subset of table rows (not necessarily aligned with user-defined partitions like date). Snowflake automatically decides how to partition data as it's loaded, typically by grouping a certain number of rows.
- For each micro-partition, Snowflake stores metadata like min and max values for each column, number of distinct values, bloom filters, etc. This metadata is used for **pruning**: at query time, Snowflake can skip micro-partitions that don't match the filter predicates (e.g., if your query filters for date=2021-01-01 and a micro-partition's date range is entirely 2020, it's skipped)datacamp.com.
- This is similar to partition pruning but at a finer granularity and automatic. It's a big reason Snowflake queries are efficient without explicit indexes – the micro-partition metadata acts like an index allowing it to only read relevant partitionsdatacamp.com.
- Micro-partitions are immutable once written; updates/deletes in Snowflake are handled by creating new micro-partitions (and marking old ones as obsolete).
- The term basically highlights Snowflake's approach to automatically partition the data for you in micro chunks and use column stats for performance.

- It's different from user partitions in other DBs – you don't specify micro-partitions, Snowflake manages it.
- Because they are small, skipping them leads to scanning minimal data. And because they are sorted by insertion, clustering (or manually defining a cluster key) can further improve how data is laid out for micro-partitions to align with query patternsvervecopilot.com.

What interviewers look for: That you know Snowflake's secret sauce. "Micro-partition" plus mention of pruning via metadata is key. Shows deeper Snowflake knowledge beyond just normal DB.

Common pitfalls:

- Confusing with Hadoop HDFS blocks – conceptually micro-partitions are different (HDFS blocks are lower-level storage; micro-partitions are logical in Snowflake).
- Not mentioning the benefit (pruning) – that's the whole point.

Follow-up: If performance on a certain large table is slow for specific queries, how might you use clustering with micro-partitions? (Expected: define a cluster key so Snowflake reorders micro-partitions by that key in maintenance, enhancing pruning for that key queries.)

---

48. **(Medium)** What is Delta Lake (the storage format) and how does it provide ACID on a data lake?

Answer:

- **Delta Lake** is an open-source storage layer that adds **ACID transactions** to Apache Spark and data lakes. Data is stored in **Parquet files** along with a **transaction log (JSON)** that tracks all changesdremio.comdremio.com.
- The transaction log (often _delta_log folder) has commit files that list data file additions and deletions. Delta Lake uses this to ensure atomicity: a write either all gets recorded in the log or not at all. Readers will only see committed transactions in the log.
- It supports **schema evolution** and enforcement, **time travel** (because old log versions can reconstruct older table states), and **concurrent writes** by serializing commits in the log (Spark jobs use optimistic concurrency and conflict detection with the log).
- ACID:
  - **Atomicity:** Either a transaction's file additions/deletions are committed in log or not. If job fails mid-way, no partial commit in log.
  - **Consistency:** The log plus files remain consistent, with optional constraints enforcement on schema.

    - **Isolation:** Concurrent transactions don't see partial results of each other; Delta uses optimistic concurrency (if conflict, one will retry/fail).
    - **Durability:** Once commit is in log and files in storage, it's permanent (the log is stored on distributed storage as well).
- So on a data lake (like S3), normally writing might be eventually consistent and no way to do row-level updates. Delta solves by writing new files and using log to mark old ones deleted. Readers read latest log snapshot to get current table state.
- Delta also implements **data versioning**: each commit has a version number. You can query a Delta table at an old version to get a "time travel" view.
- It basically turns the lake into something like a slowly updated LSM-tree style database on files, enabling features like **MERGE (upsert)**, **DELETE**, **UPDATE** with correct results, which plain Parquet on a lake couldn't do safely while others read the same data.
- The concept of lakehouse is built on things like Delta Lake providing these features.

What interviewers look for: Key words: transaction log, Parquet, ACID, upserts. They want to confirm you know Delta is not just a file format but a log + format enabling DB-like operations.

Common pitfalls:

- Just saying "Delta Lake is Parquet with some extra info" too vaguely – mention log explicitly.
- Confusing Delta Lake with Databricks platform – Delta is the format open source, though created by Databricks.
- Forgetting concurrency control: might mention that it handles concurrent writes with optimistic locking.

Follow-up: Name two other similar table formats that provide ACID on data lake. (Answer: Apache Iceberg, Apache Hudi. Possibly Hive ACID but that's older tech.)

---

49. **(Hard)** How do BigQuery's architecture and pricing influence how you should design queries?

Answer:

- **Architecture influences**:
    - BigQuery is **serverless** and uses a massive parallel query engine (Dremel). Storage is decoupled on Colossus (with Capacitor columnar format, which uses compression and can skip data)panoply.iopanoply.io.
    - Since it reads a lot of data by default, you should design queries to **minimize data scanned**. This means using **partitioned tables** (by date or other field) and **clustering** so that BigQuery can prune data it needs to readdatumo.io.
    - If you query without filters, BigQuery will scan entire table (costly and slower). So designing schema with partitioning (e.g., partition by date) will automatically cause

query to only scan relevant partitions (which reduces billed bytes).
    - BigQuery doesn't use indexes; it relies on full scans (with possible partition/clustering pruning). So queries should avoid SELECT * and super broad scans unnecessarily – better to project only needed columns and filter by partition keys etc., to reduce data read.
    - **No updates** in traditional sense (until recently, now it supports DML but historically not heavy), so often you design in append-only or use MERGE carefully (which under hood can be expensive as rewriting data).
    - The query execution is highly parallel – you don't manage that, but should avoid very complex user-defined functions or cross joins that blow up data.
- **Pricing influences**:
    - On-demand pricing charges by **bytes scanned** in queries (and bytes stored per month). So you pay for how much data your query reads.
    - Therefore, it's beneficial to **partition** tables (so querying one day doesn't scan the whole table)datumo.io.
    - Also **prune columns** – BigQuery only reads columns you select, so don't use SELECT * if not needed. Perhaps use narrower tables or separated tables if certain columns rarely needed.
    - Using **clustering** on frequently filtered columns can let BigQuery quickly eliminate blocks that don't match filter, reducing scan (less cost).
    - Materialize aggregations (via scheduled queries to smaller summary tables) if you often do heavy scans for same results, because scanning pre-aggregated small table is cheaper.
    - **Sharding vs partitioning**: BigQuery suggests partitioning instead of having many sharded tables (like table_20220101, etc.) because partitioning is easier and avoids overhead. Pricing-wise, partitioning ensures you only pay for relevant partitions.
    - If you use the flat-rate (slots) pricing, scanning cost not directly a factor but slot consumption is – still, efficient queries mean needing fewer slots/time.
- E.g., for a large fact table, partition by date and cluster by user_id if many queries filter by user, etc., so queries on a user and time range are cheap.
- Summarize: BigQuery's model encourages denormalized schemas (to avoid expensive joins, though it can handle join but then you scan both big tables) and filtering by partition, minimal columns – effectively reading less data to lower costs and speed up.

What interviewers look for: That you know BQ charges per data processed, so query design is about reducing scanned data via partitioning and good filtering. Also maybe mention not having to index but clustering helps.

Common pitfalls:

- Not mentioning pricing at all (the Q explicitly links architecture & pricing).
- Still talking as if you need to create indexes (out of context for BQ).
- Or focusing on irrelevant things like "use caching" – BQ does have a cache for repeat identical queries for short time, but main is above.

Follow-up: If you had a large normalized schema in BigQuery, would you consider denormalizing it? (Likely yes, typical best practice is to denormalize (within reason) to reduce

join cost, since BQ can handle wide tables and joins cost in scanning multiple tables.)

Sources (Warehouses & Lakehouses):

- **Snowflake docs/blogs** – micropartitions and pruningdatacamp.comblogs.rajinformatica.com
- **Databricks** – blog on Delta Lake (maybe the Dremio blog or DeltaIO docs)dremio.comdremio.com
- **BigQuery docs** – best practices: partitioning, clustering to reduce scanned datadatumo.io, architecture highlights (Dremel)panoply.io
- **Databricks Lakehouse** – probably something like Databricks blog "What is a Lakehouse" describing ACID on lake etc., concept covered above
- **Iceberg/Hudi** – could mention but not needed, focus on Delta as example
- **Snowflake vs BigQuery vs Redshift** – many blog comparisons (e.g., Panoply, or tech blogs like medium), but used knowledge mainly.
