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.

Data Modeling (3NF, Star, Vault, Medallion)

Focus: Conceptual and physical data modeling for both OLTP (normalized schemas) and OLAP (dimensional schemas). Topics include normalization forms, schema design patterns (star vs snowflake vs data vault), and modern data lakehouse layering (bronze/silver/gold a.k.a. medallion architecture). This tests your ability to structure data for different needs – ensuring integrity and optimizing for queries.

Why It Matters: Data engineers often design tables and schemas that underpin analytics. A well-designed model ensures data is consistent, performant to query, and maintainable.

Different use cases call for different modeling approaches, so understanding the trade-offs is key

---

11. **(Easy)** What is **3rd Normal Form (3NF)** in database design, and why is normalization important?

Answer:

- **Third Normal Form (3NF)** is a database schema design approach where a table is in 2NF and, additionally, **all its columns are dependent on the primary key and nothing but the primary key**datalemur.com. In other words, no transitive dependencies: non-key fields should not depend on other non-key fields.
- Example: In a table of Employees, if you store DeptName along with DeptID (and DeptName depends on DeptID, not directly on EmployeeID), that violates 3NF. The fix is to separate Department into its own table and reference by DeptID.
- Normalization (through 1NF, 2NF, 3NF, etc.) **reduces data redundancy and update anomalies**datalemur.com. It ensures each fact is stored in one place – making inserts, updates, deletes more consistent and reducing anomalies like inconsistent data.
- Importance: In OLTP systems (transactional databases), normalization helps maintain data integrity (e.g., you update a customer's info in one row of one table, not scattered in multiple places). It also usually saves storage by eliminating duplicate data.
- However, highly normalized schemas require joins to reassemble data, which can impact read query performance; hence in data warehouses often we denormalize after initial 3NF in source systems.

What interviewers look for: Understanding of normalization principles – especially the core idea of eliminating redundancy. They also might expect mention of higher normal forms preventing anomalies (insert/update/delete anomalies). A clear simple definition of 3NF is key.

Common pitfalls:

- Giving a too theoretical definition without a simple example (e.g., just saying "each non-key must depend on key" without clarifying).
- Confusing 3NF with other forms (like describing 2NF or BCNF incorrectly).
- Not mentioning *why* we normalize – the practical benefits (integrity, no duplicate update needed) – which shows the candidate knows purpose, not just definition.

Follow-up: What are potential downsides of too much normalization? (Expected: Many joins for queries, which can hurt performance in analytics; sometimes over-normalization can make querying complex, hence star schemas for read-heavy systems).

---

12. **(Easy)** Explain the difference between a **Star Schema** and a **Snowflake Schema** in data warehousing.

Answer:

- A **Star Schema** is a multi-dimensional model with a central **fact table** (containing measures like sales, clicks, etc.) connected directly to several **dimension tables** (like Date, Product, Customer)projectpro.io. The dimension tables are denormalized (flattened) and not further broken down – each forms a “point” on the star around the fact.
- A **Snowflake Schema** is a variant where dimension tables are **normalized into sub-dimensions**projectpro.io. So a dimension table might be split into multiple tables. For example, a Product dimension might snowflake into separate tables for Product Category, Product Supplier, etc., which are linked (so the diagram looks like a snowflake with branches).
- **Key differences:** In star, dimensions are denormalized (wider, redundant fields), whereas in snowflake, dimensions are normalized (more tables, less redundancy)projectpro.ioprojectpro.io. Star schemas have simpler join paths (fact to one-level dimension). Snowflake schemas require additional joins to get from fact through dimension to sub-dimension.
- **Performance:** Star schemas are generally preferred in OLAP for simplicity and fewer joins (thus often better query performance)projectpro.io. Snowflake schemas save some space and maintain a bit more data integrity in dimensions (less redundancy), but queries become more complex (more joins).
- Example: Star – FactSales (prod_id, cust_id, date_id, revenue) with dimensions Product(prod_id, name, category, ...), Customer(cust_id, name, region,...), Date(date_id, day, month,...). Snowflake – maybe Product dimension is split: Product(prod_id, name, category_id), Category(category_id, category_name) – so to get category name for a sale, you join Sales -> Product -> Category.

What interviewers look for: Clear distinction of structure and trade-offs. Possibly an understanding that star is a specific case of snowflake (a fully denormalized dimensions design). They may also want to hear that star schemas are easier for BI tools and better for performance typicallyprojectpro.ioprojectpro.io.

Common pitfalls:

- Thinking “snowflake” refers to the modern Snowflake data warehouse (no, it’s a schema design term older than that product).
- Saying star schema has no normalization but snowflake is normalized – that’s roughly true for dimensions, but fact tables in both hold foreign keys.
- Not addressing why one would use snowflake schema: typically if certain dimensions are very large and can be further normalized to reduce duplication, or for slightly easier maintenance of dimensions, at cost of performance.

Follow-up: Which schema is easier to maintain when a dimension attribute changes (like correcting a category name)? (Answer: Snowflake schema might be easier since category name is stored once in category table. In star, category name might be repeated in every product row – requiring update in many places unless you treat it dimensionally).

---

13. **(Medium)** What is a **Data Vault** model, and how does it differ from a Star Schema?

Answer:

- **Data Vault** is a modeling approach for data warehousing that is highly normalized and geared towards agility and auditability. It has three types of entities: **Hubs** (key business entities, e.g. Customer), **Links** (associative tables representing relationships between hubs, e.g. Customer-Order link), and **Satellites** (details/descriptive attributes of hubs or links, often versioned with timestamps)reddit.commatillion.com.
- In Data Vault, the idea is to separate the keys/relationships from the actual descriptive data. Hubs contain unique business keys and no redundancy, links handle many-to-many relationships, satellites store all attributes (possibly historical changes) for a hub or link.
- **Differences from Star Schema:** A star schema is denormalized for querying (dimension tables are often not normalized further), whereas a Data Vault is more normalized (hubs/links are like a third normal form core) intended for **ELT flexibility**. Star is optimized for **performance in querying**, Data Vault is optimized for **ingestion and historization** of data from multiple sources with auditability.
- Data Vault allows easy integration of multiple sources because all keys go into hubs (even if same "customer" comes from two systems, they can link to one hub), and new attributes can be added as new satellites without altering core structures. It's more **resilient to change** (good for agile developments). But it's not user-friendly for direct querying – usually, a Data Vault feeds into star schemas or cubes for end-user consumption.
- **When to use:** Large enterprise warehouses where source data is complex and changes frequently. The Data Vault forms a raw "enterprise data warehouse" layer (often called the Raw Vault), then data can be transformed into a star/snowflake (Business Vault or Information Mart) for reporting.

What interviewers look for: Awareness of Data Vault as another modeling pattern, not as widely used as star but increasingly known in enterprise DW. They expect mention of hubs/links/satellites. And an understanding that Data Vault is about **agility and historical tracking** vs star is about **performance and simplicity**.

Common pitfalls:

- Describing Data Vault incorrectly (e.g. confusing it with actual vaults or security, or mixing it up with some NoSQL concept).
- Not mentioning the components (hubs, links, satellites) – they likely want to hear those terms.

- Not understanding that Data Vault is **not for querying directly** typically – one might incorrectly say "it's better for queries" (which it isn't directly; you usually query a data mart built from it).

Follow-up: Have you ever worked with a Data Vault? How do you load a Data Vault model (e.g., when new data comes in, how are satellites updated)? This probes practical understanding: e.g., satellites often append new records with timestamps for changes (a type of SCD Type 2 handling built in), etc.

---

14. **(Medium)** In a dimensional model, what is a **Slowly Changing Dimension (SCD)** and how can it be handled?

Answer:

- A **Slowly Changing Dimension (SCD)** refers to a dimension that changes slowly over time, i.e., attributes of dimension records can change (e.g., a customer's address or a product's category) and we need a strategy to manage historical values vs current values in the data warehouse.
- Common **SCD types**:
  - **Type 1:** Overwrite the old value with new (no history kept). Example: update customer address in place. Simple but loses history.
  - **Type 2:** Keep historical records by adding a new dimension row with the change, often with effective date ranges or a current flagatlan.comatlan.com. E.g., customer moves, we close out the old record (set an end date) and insert a new record with the new address, so we retain both the old and new addresses tied to different surrogate keys.
  - **Type 3:** Add a new column to store the "old" value (limited history, e.g., previous address field in same row). Rarely used beyond one prior value because it doesn't scale for multiple changes.
- **Usage:** Type 2 is most common in analytic warehouses because it preserves full history – you can see facts associated with the dimension as they were at the time. For instance, sales fact from 2022 will link to a customer dimension record with the address at that time, whereas sales in 2023 might link to a new customer record after they moved.
- Implementation detail: For Type 2, you typically have surrogate keys on dimension (not business key) so that historical versions each get a unique surrogate. Add fields like valid_from, valid_to or an is_current flag. ETL processes detect changes by comparing incoming dimension data to the current dimension record.
- SCD management prevents data "jumping" when attributes change; it's crucial for accurate historical analyses (like analyzing revenue by sales region with the correct region definitions at that time).

What interviewers look for: Familiarity with the term SCD and understanding of at least Type 1 vs Type 2 differences. Many data engineering roles involve maintaining dimension data, so this is practical knowledge. Mentioning real-world example helps (customer address change, product category realignment, etc.).

Common pitfalls:

- Forgetting the types or mislabeling them (mixing type 1 and 2 definitions).
- Not recognizing the need for surrogate keys in Type 2 or how fact tables link (fact still links by surrogate key).
- Overlooking that Type 2 increases dimension table size over time and complicates ETL, but it's a trade-off for accuracy.

Follow-up: How would you handle a rapidly changing attribute (so-called "Rapidly Changing Dimension") – e.g., a user's last login timestamp in a dimension? (Likely answer: that might not belong in a dimension at all, maybe keep in fact or a separate mini-dimension, because updating every change as Type 2 would explode records. Or treat as Type 1 if history not needed.)

---

15. **(Medium)** Describe the **Medallion architecture (Bronze/Silver/Gold layers)** in a data lakehouse and its benefits.

Answer:

- The **Medallion architecture** is a data organization pattern popularized by Databricks for lakehouse systemsdatabricks.commedium.com. It organizes data into **three layers**:
  - **Bronze layer (Raw):** Ingested raw data, as close to the source as possible. May be semi-structured or duplicates included. Schema often close to source, no or minimal cleaning. This is the "landing zone" for all data.
  - **Silver layer (Cleansed/Conformed):** Data that has been cleaned, deduplicated, and transformed into a more structured form. Here, data from bronze is refined – errors handled, maybe joined with reference data, and conformed (ensuring consistent formats, e.g., consistent date formats, standardized keys)databricks.com. The silver layer provides an "enterprise view" of key entities with improved quality.
  - **Gold layer (Curated):** Aggregated or business-level data ready for analytics. This could be in the form of fact/dimension tables for BI, or precomputed feature tables for ML, etc. Gold is what end-user queries hit, representing polished data for specific use-cases.
- **Benefits:** This multi-hop approach **incrementally improves data quality** at each stagedatabricks.comdatabricks.com. It allows:
  - Easier debugging and auditing: you always can trace back to bronze raw if there's a discrepancy in gold.
  - Different consumers for different needs: Data scientists might grab Bronze for experimentation (because it's raw and complete), whereas business reports hit Gold for trustworthy numbers.

    - Modular pipelines: If upstream logic changes, only downstream from that layer might need recomputation, not everything end-to-end.
    - The Bronze/Silver separation often aligns with DataOps: Bronze is append-only raw logs (good for reprocessing), Silver is where data quality checks and transformations happen, Gold is optimized for fast reads (could even be in a data warehouse).
- This mirrors earlier patterns: e.g. **Staging → EDW → Data Mart** is analogous in warehousing. Medallion is specifically in lakehouse context, often implemented with tools like Delta Lake. Bronze might be raw Parquet/JSON, Silver as Delta tables with schema, Gold as Delta or even exported to warehouses.

What interviewers look for: Understanding of modern data lakehouse practices. Many companies building on S3/ADLS+Spark are adopting this layered approach. They want to see you know how to structure pipeline layers for reliability and claritydatabricks.com. Mentioning the concept of "incremental quality and refinement" is key.

Common pitfalls:

- Thinking bronze/silver/gold refers to data quality levels only (it does, but specifically implemented as physical layers in storage).
- Not explaining what kind of transformations happen between layers.
- Confusing this with user access levels – it's primarily about data refinement stages, though sometimes gold is also limited to certain users.

Follow-up: How would you implement data versioning or rollback in this architecture? (Expect: because bronze retains raw, you can re-run through silver/gold if an issue is found. Using ACID tables like Delta Lake for silver/gold helps with versioning/time travel. Also, testing happens at silver before promoting to gold.)

---

16. **(Hard)** You have a large **fact table** (billions of rows). What strategies can you use in the data model to improve query performance on this fact table?

Answer:

- **Partitioning:** Physically partition the fact table by a commonly filtered column (e.g., date, region). This means queries can skip entire partitions (e.g., only scan last month's partition) – significantly reducing I/Omedium.com. E.g., partition by date, each day's data in separate file(s).
- **Clustering/Sorting:** In warehouses like Snowflake or BigQuery, clustering (or sort keys in Redshift) on certain columns (like date or customer ID) can help prune data and improve compressiondocs.aws.amazon.com. Sorted data allows range scans on that key rather than full scans.

- **Segregate cold data:** If the fact table spans many years, consider moving old data to separate table or archive (if rarely accessed) to keep main fact smaller. Alternatively, use a data lake for older data and the warehouse for recent – but that complicates queries if needed together.
- **Aggregations / Summaries:** Create aggregated tables (roll-ups) on common groupings. If queries often ask for monthly totals or certain dimensions, maintaining a summary table can avoid scanning the base fact every time (this is more of an additional model artifact, not altering the fact but supplementing it).
- **Sharding (in distributed systems):** If using a distributed database, distribute the fact table on a key that aligns with join/filter patterns (e.g., distribute by customer ID if queries often filter by customer, to localize data).
- **Columnar storage & compression:** Ensure the fact table is in a columnar format (most warehouses do this by default). Only needed columns are read. Also storing as partitioned Parquet (if on data lake) for efficient reads.
- **Proper indexing** (if using a relational DB): On massive fact tables, full indexing is often impractical, but sometimes a covering index for a very frequent query can help. In columnar warehouses, indexing is less common, but some have search optimization features.
- **Denormalization vs. star join optimizations:** In a star schema, ensure dimension keys in fact are surrogate (integers) for small join footprint, and maybe use techniques like join elimination or pre-joining if necessary to avoid too many lookups.
- **Hardware/compute scaling:** Not a modeling change per se, but ensure you're leveraging parallel processing – e.g., in BigQuery, having enough slots, or in Spark, enough partitions. You might design the model (like partition counts) to align with parallelism.

What interviewers look for: Practical approaches to handling large data volumes. They want to know you wouldn't just let a 1B row table sit unpartitioned. Partitioning is usually the first expected answer. Also, demonstrating knowledge of both physical (storage-level) optimizations and logical (aggregate table) strategies.

Common pitfalls:

- Suggesting too many indexes on a huge fact (if it's a warehouse scenario, indexes are not usually the primary tool).
- Ignoring the specifics of the platform (solution might differ for Redshift vs BigQuery vs Hive, etc., but partitioning is common to all).
- Not mentioning partitioning at all – which is a red flag since any big data modeling should consider that.

Follow-up: If the fact table is partitioned by date, how would you handle a query that needs an **annual** summary? (Expect: the query will still scan partitions for that year, which is fine. Or if needing faster, maintain a summary table for year or ensure partition pruning works across range. Possibly mention partition elimination across multiple partitions and avoid scanning all).

17. **(Hard)** How do you model a **many-to-many relationship** in a star schema? For example, Customers and Products are many-to-many via purchases.

Answer:

- In a star schema (dimensional model), a many-to-many relationship between two business entities usually is represented via the **fact table**. The fact table's grain might be the intersection (e.g., a purchase event linking a customer and a product).
- **Example:** Customers buy Products. A naive approach might think of a direct Customer-Product bridge, but in dimensional modeling we'd have a **FactSales** with CustomerID and ProductID foreign keys (plus measures like quantity, amount). That fact implicitly handles the many-to-many: a customer with multiple product purchases will have multiple rows, and products bought by multiple customers appear in multiple rows.
- If the many-to-many needs to be treated as a dimension in itself (like a "pairing" concept), sometimes you introduce a **bridge table or a multi-valued dimension**. E.g., consider modeling "Product Bundles" that multiple customers buy – you might have a bridge table linking bundle ID to product IDs. But in classic star, we try to avoid snowflaking unless necessary.
- In some cases, you create an **aggregated fact (factless fact)** for many-to-many relationships. For instance, if analyzing which Customers are connected to which Products (no numerical measure needed), you might still create a fact table with just CustomerID, ProductID (and maybe a count).
- So, the answer: use a fact table that references both dimensions. Many-to-many relationships among dimensions are resolved by going through the fact (the fact is in the "middle" of the star).
- Additionally, if a many-to-many exists purely between two dimensions without a fact (rare in analytics – usually there's an event tying them), one could create a **bridge/association table** and treat it either as a factless fact or a snowflaked dimension. For example, modeling Students and Courses (many-to-many enrollment), the enrollment itself can be a fact table (possibly factless if just a flag), linking student dim and course dim.

What interviewers look for: Recognition that star schemas don't directly connect dimensions to each other; the fact is the connector. Possibly mention of bridge tables if you know about that concept for complex cases (e.g., a many-valued dimension like a customer has multiple hobbies – you either create a bridge or model each hobby as a separate row in a factless fact).

Common pitfalls:

- Suggesting adding a direct foreign key from one dimension to another (that snowflakes the schema; not star schema approach typically).
- Not understanding the role of fact table in linking dims.
- Overcomplicating: sometimes candidates bring up advanced bridge table concepts unnecessarily. Keep it straightforward unless pressed.

Follow-up: What is a **factless fact table** and when would you use one? (Answer: a fact table with no numeric measures, only keys – used to model events or relationships that have no measure, e.g., student-course enrollment, or attendance. It's essentially to capture many-to-many occurrences for analysis, like count of events.)

Sources (Data Modeling):

- **ProjectPro (Daivi)** – *“100 Data Modeling Interview Questions (2025)”* – covers normalization, star vs snowflakeprojectpro.ioprojectpro.io
- **Matillion Blog** – *“Data Vault vs Star Schema vs Third Normal Form”* – comparative discussionreddit.commatillion.com
- **Reddit r/dataengineering** – discussion on Data Vault vs others, notes that star is for OLAP, vault for ELT agilityreddit.com
- **Atlan 2025 Guide** – *“Apache Hudi vs Iceberg vs Delta”* (mentions evolution of table formats, indirectly relevant to medallion)atlan.comatlan.com
- **Databricks** – *Medallion Architecture* glossarydatabricks.comdatabricks.com
- **Kimball Toolkit** – Concepts of SCD Type 2 (e.g., via various online summaries, such as StackOverflow answers on SCD)atlan.comatlan.com

---