Mane Global Test

Solution by: Ali Rizvi ([rizvi\_ali@live.com](mailto:rizvi_ali@live.com))

# Section 2: Data Engineering Questions

**Question1.** What are the three main stages of an ETL pipeline, and what are some challenges you might encounter at each stage?

1. **Extract:** this involvespulling raw data from one or more source systems (databases, APIs, logs, flat files etc).  
     
   *Challenges:*
   1. Source Heterogeneity: Data might come in many forms e.g. relational DB (SQL), NoSQL (MongoDB), APIs (JSON), CSV/Excel, or streaming events (Kafka). Writing connectors or drivers that handle each source’s authentication, schema can be challenging.
   2. Latency & Scale: If the source is huge (terabytes of logs) or has high‐velocity (millions of events per second), pulling data in a reasonable window can be hard. Techniques such as parallel extraction, chunking/partitioning, or incremental change data capture can handle scale.
   3. Data Consistency: If you read from a live transactional database, you may get mid‐transaction snapshots or partial updates. Ensuring the extracted data is in a consistent window (e.g. use database snapshots, transaction logs, or timestamps) is crucial.
2. **Transform:** Involves data cleanse, filtering, and reshaping the extracted data into a form suitable for analysis or loading into the target.  
     
   *Challenges:*
   1. Data Quality & Validation: Raw data often contains missing, malformed, or inconsistent records (e.g. invalid dates, null primary keys). This requires building robust validation/cleaning rules (e.g. drop or impute missing values, enforce type conversions, flag anomalies).
   2. Business Logic Complexity: Transformations might need to apply complex business rules, like, merging multiple source tables on non‐trivial keys, calculating derived metrics (e.g. daily returns, moving averages), standardizing features across systems e.g. product/transaction codes. Challenge is to ensure the code is well‐versioned and tested, so transformations remain correct as data or rules change.
   3. Performance & Resource Constraints: Large volume transforms (e.g. joining billions of records) can be ultra‐slow if done without a focus on efficiency. Challenge here is to decide whether to use in‐memory frameworks (Pandas/Dask) versus distributed compute (Spark) versus within the database e.g. using GCP BigQuery.
3. **Load:** Refers to the process to Insert or update the transformed data into the target system (data warehouse, data lake, etc).  
     
   *Challenges:*
   1. Incremental vs. Full Loads: Full refresh of a massive table can take hours/days. Better to do incremental loads. Proves difficult to manage “upsert” logic in the target—handling late‐arriving data, deduplication, or slowly changing dimensions (SCDs).
   2. Throughput & Concurrency: If many ETL pipelines or BI users hit the same target (e.g. BigQuery), you can cause lock contention or slow writes. This can be solved using batch data into efficient bulk chunks, use partitioning/clustering, and throttle concurrency.
   3. Schema Evolution & Backward Compatibility: Targets may have evolving schemas (new columns added, data types changed). ETL needs to be designed to gracefully handle schema changes—e.g. ignore new fields you don’t need, or auto‐migrate schema before load.

**Question 2:** Describe different approaches to transferring large volumes of data between systems. Consider batch processing and streaming data methods.

1. **Batch Processing:** Moving data in discrete chunks (often on a schedule).

* **Scheduled Export‐Import**: Export tables from source using a scheduler, compress them (e.g. CSV to Gzip), and then copy to target location (e.g. S3), to run a load job. Tools such as AWS Glue, Airflow + custom Python scripts allow for this.
* **Change Data Capture (CDC) + Batching**: If the source DB supports CDC (e.g. via Kafka Connect etc), capture insert/update/delete events and store them in a staging log. Then batch‐apply those logs to the target once per hour or per day.
* **Bulk Copy / Parallel Copy**: Partition data by date range or hash key, then parallelize multiple worker processes/threads to copy each chunk

*Table 1: Pros and Cons of Batch processing*

|  |  |
| --- | --- |
| Pros | Cons |
| 1. Simpler to implement 2. Data consistency within each batch (if you snapshot correctly). 3. Lower overhead—target DW can often optimize large bulk loads. | 1. Latency (not “near real-time”). 2. Might have to handle very large files (e.g. 500 GB) that take hours to move. 3. Might overload source/target. |

1. **Streaming (Real‐Time) Processing:** Continuously transfer data records as they arrive, typically with sub‐second or second‐level latency.

* **Event‐Driven Architecture**: Application servers (microservices) emit domain‐specific events (e.g. “NewOrderCreated,” “UserUpdated”) to a streaming platform. Downstream data pipelines consume those events, transform/enrich, and write to a data store.
* **Streaming ETL Services**: Fully managed services that read from sources (e.g. DB, message queue) and write to targets in a live, continuous fashion—handling retries, schema evolution, and exactly‐once semantics.

*Table 2: Pros and Cons of Stream processing*

|  |  |
| --- | --- |
| Pros | Cons |
| 1. Low latency (seconds‐to‐minutes). 2. Better for event‐driven or microservices environments. 3. Enables near real‐time analytics (e.g. dashboards that update as orders arrive). | 1. Complexity: requires maintaining a streaming platform, windowing/reconciliation logic, and ensuring idempotency. 2. Harder to debug/replay (compared to a simple daily CSV). |

**Question 3:** What are some common data transformation techniques and tools you have implemented in past projects, and how do you decide which technique to apply? Can you describe some challenges you faced in those projects?

**A. Transformation Techniques**

1. **Filtering & Cleansing:** Remove rows with missing or invalid values, enforce type conversions. Always as a first step to ensure data quality and obtain consistend results.
2. **Normalization & Standardization:** Scale numeric columns (e.g. min‐max or z‐score) or convert categorical columns to consistent codes. Essential for ML models and like-for-like comparisons.
3. **Aggregation & Summarization**: Compute daily/weekly/monthly aggregates (sum, count, avg) from more granular data (e.g. time series transaction‐level data). This either depends the project requirements and business ask or could be due to matching the data cadence of different data sets to be analysed together.
4. **Window Functions & Time‐Based Features**: Compute moving averages, rolling sums, lead/lag features (e.g. last‐quarter revenue, year‐over‐year change). This is primarily done to view PCP comparisons, change in data or to remove anomalies or noise in data that is too volatile.
5. **Joins & Lookups**: Enrich tables by joining additional data to add more depth and dimensions to the analysis. This is mainly done to give transactional data business context.
6. **Pivot / Unpivot (Melting):** Convert between “wide” and “long” formats. This is primarily done for ease of understanding or meeting conditions for different plotting techniques. This is also required if we need to match schema of another data source we are trying to join with.
7. **Type Casting & Schema Enforcement**: Enforce numeric types, string lengths, decimal precision, date formats, or enumerations. This is to avoid data type issues and analysis returning unexpected results e.g. representing percentages e.g. 95% as 95.0 or 0.95**.**

**B. Choosing a Technique & Tool**

* **Small to Medium Data (< a few tens of millions of rows):** Pandas can handle up to 10 million rows comfortably (depending on memory).
* **Large Data (10M+ rows)**: prefer to use either Apache Spark or try to push as many transformations within our cloud databases e.g. BigQuery in GCP prior to extracting the data for further analysis.

**C. Common Challenges & How I’ve Addressed Them**

1. **Memory Errors in Pandas**: Trying to join two row tables with multiple million records in a single Pandas DataFrame leads to OOM. I’ve tried to break these into smaller chunks (e.g. date‐partitioned subsets), process iteratively, and write intermediate results to disk or a database. Or when necessary, migrate the job into Spark.
2. **Schema Evolution Issues**: A source table adding/subtracting columns resulting in the downstream ETL script to break. Primary ways I’ve solved this is by either ingesting only the required columns or ignoring extras. Or by using automated test/logic to detect schema changes.
3. **Managing Slowly Changing Dimensions:** this is when information updates, or changes over time for history as well.The main requirement here is the need to keep history instead of overwriting.

**Question 4:** How do ETL (Extract, Transform, Load) and ELT (Extract, Load, Transform) differ? In what scenarios would you choose one method over the other?

**ETL** (Extract → Transform → Load): Extracts raw data from sources, transforms the data in an intermediate layers (python script, ETL server) and then loads it into target for usage.

**ELT** (Extract → Load → Transform): Extracts raw data from sources, loads it directly into the data warehouse, data lake, or analytics DB, and then transforms it inside the data warehouse (via SQL, stored procedures, etc).

*Table 3: Choosing whether to use ETL or ELT*

|  |  |
| --- | --- |
| *ETL is preferred when* | *ELT is preferred when* |
| 1. When the destination has limited compute or you need to enforce strict data quality before it touches the warehouse. 2. When there are existing ETL tools (Informatica, Talend) and want to leverage their transformation features. | 1. Your warehouse (Snowflake, BigQuery, Redshift) can handle large-scale transformations more efficiently than an external cluster. 2. You want to retain raw data in the warehouse for lineage and reprocessing (“bronze → silver → gold” layers). |

**Question 5:** Describe an experience where your pipeline was underperforming or causing system bottlenecks. What steps did you take to identify the performance issues, and what optimizations did you implement?

**Context:**  
I needed to join over 20 million web‐usage records with approximately 7 million mobile‐app usage records to identify which customers visited specific websites after performing certain actions in the app over an analysis period of 3 month data. The web data was on‐premises, while the mobile‐app data already resided in the cloud, so both datasets had to be brought together for analysis.

**Issue:**  
The initial extraction and join process repeatedly hit server limits and failed due to the sheer volume of data.

**Diagnosis Steps:**

I instrumented the Python scripts to log execution time and memory usage, pinpointing two bottlenecks:

* Joining the app and web datasets in one large pass consumed excessive memory.
* Pulling all 50 million web records into the cloud in one go exceeded network and database limits.

**Optimizations:**

1. **Batch—Monthly Chunks:** Rather than loading the entire three-month window at once, I partitioned the web data by month. Each monthly subset was extracted, transformed, and loaded (ETL) separately, staying within server thresholds.
2. **Spark Parallelization & Pruning:** Before any cloud upload, I used Apache Spark to clean and filter each monthly chunk—removing irrelevant columns and rows—so that only the minimum required fields (“customer identifier,” “timestamp,” “event type”) were sent to the cloud. This reduced each batch’s size by roughly 70 percent.
3. **Unified Hash Key:** To match customers across web and app data without repeated expensive string comparisons (phone number, email, etc.), I generated a single hash key for each customer based on a combination. Both Spark jobs and the cloud‐side database used this hash, so each join became a simple integer lookup instead of multiple field comparisons.

**Result:**

* **In‐Spark Aggregation:** Each monthly web chunk was cleaned, hashed, and aggregated in Spark—yielding a much smaller summary table.
* **Batched Cloud Load:** Those reduced monthly tables were uploaded sequentially to the cloud database without exceeding limits.
* **Final Join & Analysis:** In the cloud, I executed a single SQL join between the precomputed web‐events tables and the 7+ million-row app usage table (both keyed on the unified hash). This let me efficiently identify which customers visited target websites after app actions.
* **Outcome:** The entire three-month join and analysis was completed and never triggered server errors.

**Question 6:** How do you or have you managed environment creation and Python package management currently or previously in your role?

Depending on where the project was done, there were two approaches that I’ve usually followed

1. **Larger Projects:** We leveraged Cloudera Data Science Workbench at my organisation, which provisions isolated Conda environments on Hadoop clusters. Each project got its own virtual environment, preventing dependency conflicts and ensuring reproducible jobs on distributed data.
2. **Small Proof-of-Concepts or Local Development:** I use Python’s built-in venv (or virtualenv) to create lightweight, isolated environments. Each project maintains a requirements.txt for pip install so anyone can reproduce the exact package setup.