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.

Batch Processing (Spark, Hadoop, EMR, Dataproc)

Focus: Large-scale batch data processing frameworks and concepts. Emphasis on Apache Spark (by far the most asked-about tool) – covering RDD vs DataFrame, transformations vs actions, shuffle, partitioning, memory management, etc. Also touches on Hadoop ecosystem (HDFS, MapReduce concepts) and running Spark on clusters (YARN, standalone, cloud managed services like EMR or Dataproc). Expect scenarios of optimizing batch jobs, handling skew, understanding execution model.

Why It Matters: Batch ETL jobs form a core part of data pipelines. Efficient use of frameworks like Spark can mean the difference between a job running in 2 hours vs 10 hours. Interviewers test if you grasp distributed processing basics and can solve problems like data skew or slow joins.

---

18. **(Easy)** What is Apache **Spark** and how is it different from Hadoop **MapReduce**?

Answer:

- Apache **Spark** is a unified analytics engine for large-scale data processing. It provides in-memory computing capabilities, high-level APIs in multiple languages (Scala, Python, Java, R), and supports batch, streaming, SQL, machine learning, and graph processing in one frameworkdatacamp.comdatacamp.com.
- The main difference from classic Hadoop MapReduce:
  - **Speed:** Spark performs many operations **in-memory**, reducing disk I/O. MapReduce writes to disk between each map and reduce step. Spark can be orders of magnitude faster for iterative algorithms or multi-step pipelines because it keeps data in RAM across stepsdatacamp.com.
  - **Ease of Use:** Spark has easier APIs (DataFrames, SQL, functional RDD APIs) vs. writing Java MapReduce jobs which is more verbose. Also, Spark comes with interactive shells.
  - **General Engine:** MapReduce is primarily a batch processing engine. Spark has libraries for streaming (Structured Streaming), machine learning (MLlib), graph (GraphX), etc., all integrated.
  - **Execution Model:** MapReduce strictly has map phase, shuffle, reduce phase. Spark builds a DAG of transformations and optimizes the whole pipeline, executing with many possible operators beyond just map and reduce (e.g., joins, filters as separate stages).
  - Spark can run on top of Hadoop (using HDFS for storage and YARN for resource management) or standalone or in the cloud. MapReduce is tied to Hadoop.
- In summary: Spark is a faster, more flexible, in-memory data processing engine compared to the older MapReduce paradigm, which was disk-based and more limited in scope.

What interviewers look for: A concise understanding of Spark's value prop (speed via in-memory, versatility) and how it evolved from Hadoop era. Possibly expecting mention that Spark can leverage Hadoop ecosystem (not an either/or necessarily, Spark often runs on Hadoop clusters).

Common pitfalls:

- Claiming “Spark always in-memory, never uses disk” – it spills to disk if data doesn’t fit or for big shuffles, but it *tries* to use memory more.
- Not knowing that Spark can work with YARN/HDFS – some think Spark replaced Hadoop entirely; Spark often complements Hadoop.
- Forgetting to mention Spark’s multi-module nature (Spark SQL, etc.) – this highlights how it’s more than just a MapReduce replacement.

Follow-up: What is one disadvantage of Spark vs MapReduce? (Could mention memory usage – Spark needs lots of RAM, whereas MapReduce can work with disk for everything, making Spark possibly expensive; also Spark’s complexity in tuning can be higher, etc.)

---

19. **(Easy)** In Spark, what is the difference between a **transformation** and an **action**? Give examples.

Answer:

- In Spark’s RDD/DataFrame API, a **transformation** is an operation that defines a new dataset from an existing one, but is **lazy** – it doesn’t execute immediately. It builds up the lineage (DAG) of computations. Examples: map, filter, select, join, groupBy are transformations. They return a new RDD/DataFrame.
- An **action** actually triggers execution of the transformations to produce a result or side effect. Actions materialize the result either by returning it to the driver or writing it out. Examples: collect(), count(), take(), show(), write/save (writing DataFrame to storage) are actions.
- Spark uses lazy evaluation: transformations are recorded, but computation only happens when an action requires a result. This allows Spark to optimize the execution plan (e.g., pipeline operations, reorder filters, etc.) before running.
- Example: If you do val rdd2 = rdd1.filter(...).map(...) – that’s two transformations (filter, map). Spark hasn’t done anything yet. When you call rdd2.count(), Spark will then launch a job to apply the filter and map and then count the elements.
- Without an action, you can chain many transformations and Spark will not process data until needed. This is key for performance (it can collapse narrow transformations into one stage, etc.)datacamp.com.

What interviewers look for: Clarity on lazy vs eager execution. This is fundamental to using Spark correctly (e.g., to know why calling too many collects is bad, or why nothing happens until an action). Also expecting common examples as above.

Common pitfalls:

- Saying transformations “transform data in memory” – not until an action triggers them.
- Forgetting that writing data (df.write.format(...).save()) is an action (effectively).

- A subtle one: Some might say “action triggers a job” – which is true; transformations don’t trigger job on their own. Good to mention.

Follow-up: Why is lazy evaluation beneficial in Spark? (Answer: allows optimization across the whole plan, reduces unnecessary computations e.g., if you map then filter, Spark might push filter earlier; also if an intermediate result isn’t used, it’s never computed.)

---

20. **(Medium)** Explain **narrow vs wide transformations** in Spark and why the distinction is important.

Answer:

- A **narrow transformation** is one where each output partition depends only on a small subset of data from one input partition – there’s a one-to-one or limited scope relationshipdatacamp.com. Examples: map, filter on an RDD – each partition of the output RDD corresponds to exactly one partition of the parent RDD. No data movement across the network is required for narrow transformations.
- A **wide transformation** (also called shuffle transformation) is one where output partitions depend on data from **multiple input partitions** – requiring a shuffle across the cluster. Examples: groupByKey, reduceByKey, join, distinct – these require redistributing data by key or some grouping, which involves sending data over the network and writing to disk (at least temporarily).
- The distinction matters because **wide transformations incur a shuffle**, which is an expensive operation (network and disk I/O). Narrow transformations can be pipelined and executed within one stage in Spark, whereas a wide transformation marks a stage boundary in the DAGadaface.com.
- Spark’s scheduler will optimize narrow transformations by combining them into one task pipeline. When a wide transformation occurs, Spark must finish all tasks of previous stage, redistribute data (hash or sort by keys), then start a new stage for subsequent tasksadaface.com.
- Recognizing these helps in optimization: e.g., using mapPartitions (narrow) vs overly doing groupBy (wide). Or using reduceByKey (which does map-side combine, still a shuffle but more efficient than groupByKey).
- Example: If you do rdd.map().filter().flatMap() – all are narrow, Spark can do them all in one go per partition. But if next you do groupByKey(), that’s wide – Spark will shuffle all data by key to group values, which is much heavier.

What interviewers look for: Understanding of Spark execution model (stage splitting, shuffle)datacamp.com. Knowing examples of each kind. It’s fundamental to reasoning about performance in Spark.

Common pitfalls:

- Thinking of it in terms of "transformations on one machine vs many" – not exactly; the key is data movement.
- Not realizing join is wide (because keys from many partitions need to come together).
- Not mentioning the performance impact: narrow is cheap, wide is costly (sometimes candidates just define but don't say why it matters – the "so what").

Follow-up: How does Spark minimize the cost when performing wide transformations? (Expect mention of things like **combiner** for reduceByKey (map-side partial aggregation), **sort-based shuffle improvements** in newer Spark, **partitioning** to avoid shuffle if upstream RDD was partitioned by same key, and **Adaptive Query Execution** possibly adjusting shuffle partitions).

---

21. **(Medium)** What is a **shuffle** in Spark and when does it occur? Why can shuffles be problematic?

Answer:

- A **shuffle** in Spark is the process of redistributing data across partitions, typically across the network, between stages. It occurs when an operation (wide transformation) requires data to be grouped or rearranged by key or some grouping that can't be done within the original partition.
- Examples of operations causing shuffle: groupByKey, reduceByKey, join, distinct, repartition, coalesce (when increasing partitions or shuffle option), sortBy, and any *ByKey that aggregates by key. In Spark SQL, operations like GROUP BY or JOIN on non-partitioned keys cause shuffles.
- During a shuffle, Spark writes the data from each map task to disk in sorted buckets (or partitions) and then those buckets are sent to the appropriate reducers (tasks in next stage)adaface.com. This involves disk I/O and network I/O.
- **Why problematic:** Shuffles are expensive – they incur heavy disk and network usage, and also require coordination (a barrier between stages). They also use a lot of memory for buffering data, and if not handled can lead to **OutOfMemory** or long GC pauses if shuffle blocks are big.
- Shuffles can also lead to **data skew** issues – if one key has a huge amount of data, one reduce task might get overloaded (straggler) while others finish quickly. This imbalanced shuffle causes slow stage completion.
- Also, shuffle files on disk consume storage and if the job fails mid-way, those interim files accumulate (Spark usually cleans up shuffle data when context ends, but large shuffles can stress the IO subsystem).
- Spark's performance tuning heavily focuses on reducing unnecessary shuffles or optimizing them (like using broadcast join to avoid shuffle of a small table, or controlling spark.sql.shuffle.partitions to a reasonable number for parallelism).

What interviewers look for: Understanding of the cost and implications of shuffles. That the candidate knows to avoid them when possible or handle them (like partitioning data beforehand). Also, being aware of skew and network overhead as reasons to be careful with shuffles.

Common pitfalls:

- Vague answer like "shuffle is mixing data around" without clarity on why or how.
- Not mentioning at least an example that triggers it.
- Not understanding that shuffle is inter-stage (some might think any data movement in cluster is shuffle; e.g. map tasks reading HDFS blocks is not a shuffle, that's just input read).

Follow-up: If you have a dataset with a highly skewed key (e.g., one key accounts for 50% of data) and you do a reduceByKey, what approaches can mitigate the skew? (Possible answers: custom partitioner that splits that key into multiple partitions artificially; or do a two-phase reduce – pre-partition data adding a salt to keys; or use map-side partial aggregation if possible; or increase parallelism for that key by splitting it).

---

22. **(Medium)** Spark offers **persist()/cache()** on RDD/DataFrame. When would you use it, and what storage levels are available?

Answer:

- **persist()/cache():** You use these to **materialize and reuse** an RDD or DataFrame in memory (or disk) across multiple actions. Spark's lazy evaluation means if you reuse the same dataset in multiple actions, by default it will recompute from scratch each time. Caching avoids that by storing the data after the first computation.
- Use case: If you have an expensive intermediate result that is needed by multiple downstream computations, you cache it so subsequent actions can reuse it quickly. E.g., you filter a large dataset to a smaller one, and then run several aggregations on that filtered set – caching the filtered set means the expensive filter is done once.
- The typical call is df.cache() which is shorthand for persist(StorageLevel.MEMORY_ONLY) in Spark. This attempts to keep the data in memory.
- Storage levels:
  - **MEMORY_ONLY:** (default) store as deserialized objects in the JVM heap. If it doesn't fit, drop some partitions (recompute when needed).
  - **MEMORY_AND_DISK:** store in memory as much as fits, and spill the rest to disk.
  - **MEMORY_ONLY_SER:** store serialized (compressed) objects in memory (saves space but adds CPU to deserialize on use).
  - **MEMORY_AND_DISK_SER:** combination of above – if not enough memory, spill to disk, all in serialized form.

    - **DISK_ONLY:** skip memory, write cache to disk only.
    - **OFF_HEAP (experimental):** store in off-heap memory (requires configured off-heap).
    - Also variants with "_2" etc. for replication across nodes.
- You'd choose storage level based on data size and speed trade-offs. MEMORY_ONLY is fastest if you have RAM for it. If data is huge, MEMORY_AND_DISK ensures it still persists without OOM, at cost of some disk I/O.
- Remember to call unpersist() if needed to free memory when that dataset no longer needed.
- Without caching, Spark recomputes lineage. Caching too much or the wrong data can waste memory, so use it when reuse justifies it.

What interviewers look for: That you know caching in Spark is explicit (unlike some systems with automatic caching). Knowledge of the common storage levels (Memory only vs memory+disk) shows experience. Also understanding of when it's beneficial.

Common pitfalls:

- Saying "Spark automatically caches data frames" (no, not unless you call cache or do repeated SQL queries maybe in Spark UI it might cache some metadata, but not actual data).
- Not knowing that cache by default is memory-only.
- Failing to mention that if data doesn't fit in memory in MEMORY_ONLY, Spark will recompute partitions as needed (so might not be fully cached).

Follow-up: If you call df.cache() and then do df.count() and then df.collect(), how many times is the data computed? (Answer: The first action (count) will trigger compute and cache the data. The second action (collect) will use the cached data, so each partition won't be recomputed, it will be read from cache. Without cache, both actions would compute the whole plan separately.)

---

23. **(Medium)** How would you approach **tuning a join** in Spark if one of the tables is very large and the other is small?

Answer:

- For a very large table and a small table, a classic optimization is to use a **Broadcast Join**. Spark's broadcast join (in DataFrame API, it's broadcast(smallDF) or automatically done if the planner thinks it's beneficial) will send the small table to all executors in memorygithub.com. Then each partition of the big table can join with the small data locally, avoiding a shuffle of the big table.
- This turns a potential shuffle join (which would shuffle both sides by join key) into a **map-side join**, where only the small dataset is replicated (which is fine if it's indeed small enough to broadcast).

- Ensure the small table is under Spark's broadcast threshold (default ~10MB, configurable via spark.sql.autoBroadcastJoinThreshold). If it's slightly above and still feasible, you can raise the threshold or explicitly hint to broadcast.
- If broadcasting isn't possible (both sides large, or one side medium but not tiny), then:
  - Make sure join keys are partitioned appropriately. Possibly **repartition** the large table by the join key ahead of the join so that shuffle happens in a controlled way (though Spark will handle this typically in the join anyway).
  - Ensure data skew isn't an issue: if one key is extremely common, that partition becomes a hotspot – might consider techniques like key salting.
  - Choose the right join type (avoid cartesian joins unless necessary).
- Also consider the format: if using RDDs, maybe use reduceByKey or cogroup for certain patterns. But in DataFrame/Dataset, focus on broadcast or sort-merge join adjustments.
- Example scenario: Joining a dimension lookup (small) to a huge fact – definitely broadcast the dimension.
- Also check if the join can be pushed down or done differently (e.g., maybe do a map lookup by collecting small data as a map on driver and using a UDF – viable if truly small and if Spark's broadcast overhead is not ideal for some reason).

What interviewers look for: Knowing about Spark's broadcast join mechanism is keygithub.com. Many DEs have used SparkSQL, so this is common knowledge expected. Also demonstrating not to blindly shuffle big data if avoidable.

Common pitfalls:

- Not mentioning broadcast at all (would seem like lack of knowledge of Spark performance features).
- Trying to broadcast a huge table (must emphasize one side should be small).
- Overlooking memory impact: broadcasting a 500MB table will likely cause issues, so "small" really means small.

Follow-up: How does Spark decide which join method to use by default? (Answer: In Spark SQL, if a table is below the autoBroadcastJoinThreshold, it will broadcast by default. Otherwise it might choose sort-merge join if data is sortable, or shuffle-hash join for certain cases. In RDD API, joins always cause shuffle unless you manage pairing or broadcast manually.)

---

24. **(Hard)** What are **Spark partitions** and how does the number of partitions affect performance?

Answer:

- A **partition** in Spark is the basic unit of parallelism – essentially a chunk of the dataset (RDD/DataFrame) that is processed by one task. Each RDD/DataFrame consists of multiple

partitions, distributed across the cluster.

- The number of partitions determines how many concurrent tasks can execute for that dataset. More partitions = potentially more parallelism (up to number of cores in cluster) but also overhead in scheduling and smaller tasks.
- If partitions are too **few** (e.g., one big partition for 1TB of data), then you won't utilize cluster parallelism well – many cores will sit idle and one executor will do all the work (and might even OOM because partition too large).
- If partitions are too **many** (e.g., millions of tiny partitions), overhead of task scheduling and coordination can dominate, and you may spend more time scheduling than computing. Also memory overhead for each partition's metadata.
- Spark's default partition count often comes from the data source (e.g., number of HDFS blocks for an RDD from a file) or defaults like spark.sql.shuffle.partitions (which defaults 200 for shuffle operations).
- You can tune partitions using repartition() (which does a shuffle to redistribute data into a given number of partitions) or coalesce() (which can reduce partitions without full shuffle, if shrinking).
- A good practice: Have at least as many partitions as total CPU cores in cluster (to utilize all). Sometimes 2-4x cores to allow better load balancing. E.g., if 50 cores, maybe 200 partitions for large data shuffle is okay (Spark default 200).
- Also consider data size: aim for partitions of reasonable size (commonly tens of MBs to maybe a few hundred MB in memory). If partitions are multiple GB, that's too large for one task's memory.
- Performance is impacted by skew too: if one partition holds much more data than others, you get imbalanced tasks (some tasks finish quick, one drags). So partitioning strategy (like key-based partitioning for joins) should aim to evenly distribute data.

What interviewers look for: Understanding of partitioning's role in parallel processing and the trade-off of number of partitions (not just "more is better"). Possibly expecting mention of Spark's default and ability to change it.

Common pitfalls:

- Talking about "partitions" in a database sense or other context – need to clarify it's Spark's in-memory/workload partition, not like a Hive partition (though related concept but not the same usage here).
- Not addressing what happens if partitions badly chosen (like OOM or idle CPU).
- Failing to mention how to adjust partitions.

Follow-up: If you notice one task is running much slower than others in a stage (skew), how could you mitigate that partition skew? (Answer: Maybe repartition by a different key or add a random salt to distribute that heavy key, or break the data in that partition further. Also could manually identify the skewed key and handle it separately.)

---

25. **(Hard)** You run a Spark job and notice a **stage with a very long task (straggler)**, while others finish quickly. What could be the cause and how to address it?

Answer:

- A single long-running task in a stage typically indicates **data skew** – one partition has a disproportionate amount of data or a particularly expensive chunk. For example, if doing a groupByKey on a dataset where one key has far more records than others, the reducer handling that key will run much longer.
- Other causes: Maybe a non-deterministic partitioner or a custom partitioner bug causing uneven distribution. Or one executor node is slower (perhaps dealing with hardware issues or GC pause).
- If it's skew: Solutions include:
  - **Salting the key:** For the skewed key, split its data across multiple fake keys. E.g., instead of key "XYZ" for all, assign "XYZ_1", "XYZ_2", etc. in the map side, then later combine. This distributes that one key's data across multiple tasks. It adds complexity in reducing but helps parallelize.
  - **Increase partitions** for that stage (increase spark.sql.shuffle.partitions or using repartition() before the heavy operation) so that the large partition might be split into two or more (Spark by default partitioner might not split one key across partitions, but if multiple keys were together in one partition, more partitions might break them up).
  - If using Spark SQL, there's a feature in modern Spark called **Adaptive Query Execution (AQE)** that can automatically detect skew and split the skewed partitions into smaller ones in shuffle join scenarios[python.plainenglish.io](python.plainenglish.io). Ensuring AQE is on (spark.sql.adaptive.enabled=true) might help.
  - If one node is slow (not skew but environment): maybe the executor has too high GC or is spilled to disk. Check if the straggler is consistently on same node – could be GC tuning issue or need to increase executor memory or fix data spilling. If it's GC, consider more partitions (smaller tasks) or increasing heap or using memory-efficient data structures.
- Could also consider **broadcast join** if applicable (if skew from join side).
- Summarizing: Identify if skew by looking at stage's task metrics (one task data size much larger). Then mitigate by repartitioning or salting to redistribute load.

What interviewers look for: Troubleshooting mindset. Recognizing skew as a common Spark performance issue. Familiarity with techniques like salting (which is a known workaround). Possibly mention of Spark's adaptive execution if they expect knowledge of new features.

Common pitfalls:

- Answering generically "maybe the executor is slow" but not addressing that in distributed data, skew is most common cause for one slow task.
- Not providing a concrete solution beyond "increase resources" (which might not fix skew because one partition would still be big).
- Forgetting that maybe an UDF or something in that one partition could be hanging – but if systematically one task, likely data distribution.

Follow-up: How would you detect skew in a Spark application? (Answer: Spark UI – look at Stage detail, tasks summary. If one task has way more input data or runtime than others, that's skew. Also looking at data distribution by key counts if possible.)

Sources (Batch Processing & Spark):

- **DataCamp Spark Q&A (2024)** – transformations vs actions, Spark basicsdatacamp.comdatacamp.com
- **Medium (Sanjay PHD)** – Spark internals: mentions stage boundaries at wide transformationsadaface.com
- **Adaface Spark Interview** – narrow vs wide definitionadaface.com
- **GeeksforGeeks 2025 Kafka** (some Spark context in difference of streaming vs batch) – not directly used
- **Databricks Blogs** – Adaptive Execution for skew handlingpython.plainenglish.io
- **Official Spark Documentation** – on persistence storage levels (Memory_only, etc.) and shuffle mechanics (Spark programming guide)
- **StackOverflow** – common Qs on broadcast join and skew solutions in Sparkgithub.com

---
