Frankline Florence

TREDENCE DATA ENGINEERING Q&A

2025

www.linkedin.com/in/frankline-florence

TREDENCE DATA ENGINEERING INTERVIEW QUESTIONS AND ANSWERS- 2025

⚙️ Transformations & Logic

1. What's the difference between groupByKey() and reduceByKey()?
2. Can you walk through a DAG for a job that uses groupBy + join + withColumn?
3. What causes a shuffle? How can you identify and reduce it?
4. Why is coalesce() preferred over repartition() before writing data?

💾 Delta Lake + Merge Strategy

5. How would you handle schema evolution in a Delta table with streaming input?
6. What happens when two merge operations run on the same Delta table?
7. How do you implement audit columns (created_dt, updated_dt) in Delta merge logic?
8. What is the role of the _delta_log and how is ACID enforced?

🧠 Memory Tuning & Debugging

9. Your job is failing with "GC Overhead Limit Exceeded" — what steps will you take?
10. How do you monitor task skew in Spark UI and fix it?
11. How does persist() differ from cache() — and which is safer for streaming pipelines?
12. What's the impact of setting spark.sql.shuffle.partitions too high?

🌊 Streaming Scenarios

13. How do you design a watermark strategy for late-arriving Kafka events?
14. What's the difference between trigger(once) and trigger(processingTime)?
15. How do you recover from checkpoint corruption in structured streaming?

ANSWERS

⚙️ TRANSFORMATIONS AND LOGIC

🟧 **What's the difference between `groupByKey()` and `reduceByKey()`?**

- `groupByKey()`:
  - Groups all values with the same key into a **single list** on the same partition.
  - Causes **heavy data shuffling**, which is inefficient for large datasets.
  - Not preferred for aggregations due to its memory overhead.
- `reduceByKey()`:
  - Performs **local aggregation** (on each partition) **before** shuffling.
  - More efficient and **saves network I/O**.
  - Recommended for large-scale aggregations.

➡️ **Use `reduceByKey()` for aggregations whenever possible.**

🟧 **Can you walk through a DAG for a job that uses `groupBy` + `join` + `withColumn`?**

1. **`groupBy`:**
   - Triggers a shuffle to group records by key across partitions.
   - Outputs an intermediate logical plan node.
2. **`join`:**
   - If not a broadcast join, also triggers a shuffle on both sides.
   - Depending on the join type (e.g., inner, left), the DAG will have additional stages for preparing and joining datasets.

3. `withColumn`:
    - A narrow transformation; adds or transforms columns.
    - Does not trigger a shuffle.

DAG Summary:

```
Read -> Shuffle (groupBy) -> Shuffle (join) -> Narrow
(withColumn) -> Write
```

🟧 What causes a shuffle? How can you identify and reduce it?

Shuffle occurs when:

- Data moves across partitions or executors.
- Common causes:
    - `groupBy`, `join`, `distinct`, `repartition`, `coalesce` (in some cases), `sort`.

How to identify:

- In Spark UI → DAG or Stage view → Look for "Exchange" nodes or long shuffle read/write times.

How to reduce:

- Use `reduceByKey` instead of `groupByKey`.
- Use broadcast joins for smaller tables.
- Avoid unnecessary `repartition()`.
- Use partition pruning and bucketing where applicable.

🟧 Why is `coalesce()` preferred over `repartition()` before writing data?

`coalesce(n)`:

- Reduces number of partitions without **shuffle**.

    - Merges adjacent partitions.
    - Efficient for **narrow transformations** like write optimization.

`repartition(n)`:

    - **Shuffles data** to create evenly sized partitions.
    - Expensive and time-consuming.

Use Case:

- Before writing output (like to S3 or HDFS), use `coalesce()` to reduce small files and optimize I/O.

💾 DELTA LAKE + MERGE STRATEGY

🟧 **How would you handle schema evolution in a Delta table with streaming input?**

- Enable **schema evolution options**:

```
.option("mergeSchema", "true")  # For batch

.option("cloudFiles.schemaEvolutionMode", "rescue")  # For Auto Loader
```

- For structured streaming:

Use **Auto Loader (cloudFiles)** or set `spark.databricks.delta.schema.autoMerge.enabled = true`.

Best Practices:

- Handle schema changes explicitly.
- Use **schema enforcement** + evolution cautiously to avoid silent data corruption.

🟧 **What happens when two merge operations run on the same Delta table?**

- Delta Lake uses **optimistic concurrency control**:
  - Both merges will read the same snapshot.
  - First write commits; the second fails with a **conflict**.
  - The second must **retry** using the new snapshot.

Best Practice:

- Serialize merge jobs or **use isolationLevel** with `MERGE` to avoid write conflicts.

🟧 How do you implement audit columns (`created_dt`, `updated_dt`) in Delta merge logic?

Use conditional logic in **`MERGE`** statement:

In SQL:

```
MERGE INTO target t
USING source s
ON t.id = s.id
WHEN MATCHED THEN UPDATE SET
  t.name = s.name,
  t.updated_dt = current_timestamp()
WHEN NOT MATCHED THEN INSERT *
  VALUES (s.id, s.name, current_timestamp(), current_timestamp())
```

In PySpark:

```
merge_condition = "t.id = s.id"
deltaTable.alias("t").merge(
    sourceDF.alias("s"),
    merge_condition
).whenMatchedUpdate(set={
    "name": "s.name",
    "updated_dt": "current_timestamp()"
}).whenNotMatchedInsert(values={
    "id": "s.id",
    "name": "s.name",
    "created_dt": "current_timestamp()",
    "updated_dt": "current_timestamp()"
}).execute()
```

🟧 What is the role of the _delta_log and how is ACID enforced?

- `_delta_log/`:
  - Stores Delta Lake **transaction logs** as JSON + parquet files.
  - Each file represents a commit.
- **ACID is enforced** through:
  - **Atomic commits**: Files written first, then log is committed.
  - **Isolation**: Snapshots are consistent; readers never see partial writes.
  - **Concurrency control**: Writes use optimistic concurrency with conflict detection.
  - **Durability**: Log files can rebuild table state.

🧠 MEMORY TUNING AND DEBUGGING

🟧 Your job is failing with "GC Overhead Limit Exceeded" — what steps will you take?

Cause:

- JVM spends too much time in garbage collection (GC) with little memory reclaimed.

Fixes:

1. **Increase executor memory**:

   ```
   --executor-memory 4g
   ```

2. **Reduce data per executor**:
   - Use `repartition()`, filter early.
3. **Use Kryo serializer**:

   ```
   spark.conf.set("spark.serializer",
   "org.apache.spark.serializer.KryoSerializer")
   ```

4. **Persist selectively** and **avoid caching large datasets unnecessarily**.
5. **Tune GC settings** (if using JVM directly).
6. **Use broadcast joins** or **DataFrame optimizations**.

🟧 How do you monitor task skew in Spark UI and fix it?

Detect Skew:

- Spark UI → Stages tab → Look for tasks with:
  - Very long execution time.
  - Large input size compared to others.

Fixes:

- **Salting keys**: Add random prefixes to skewed keys.
- **Broadcast small tables** to avoid join shuffle.
- Use **skew hints** in Spark 3.x:

  ```
  SELECT /*+ SKew('key') */ ...
  ```
- Use **adaptive execution** (enabled by default in Spark 3+).

🟧 **How does `persist()` differ from `cache()` — and which is safer for streaming pipelines?**

`cache()`:

- Shortcut for `persist(StorageLevel.MEMORY_AND_DISK)`.

`persist(level)`:

- Offers **fine-grained control**: MEMORY_ONLY, MEMORY_AND_DISK, DISK_ONLY, etc.

In Streaming:

- **`persist()`** with **MEMORY_AND_DISK_SER** or `DISK_ONLY` is **safer**:
  - Prevents OOM (Out Of Memory) errors in long-running jobs.
  - Avoids GC pressure.

🟧 **What's the impact of setting `spark.sql.shuffle.partitions` too high?**

<u>**Too many partitions:**</u>

- More **task overhead** (scheduler burden).
- High **disk I/O** and **small files**.
- **Increased shuffle latency**.

<u>Too few:</u>

- May **underutilize cluster resources**.
- Large partitions = **skewed tasks**, OOM errors.

Best Practice:

- Tune it based on data size:
    - 200–500 for large jobs.
    - Use `df.rdd.getNumPartitions()` to inspect.
    - Use **adaptive execution** to dynamically tune at runtime.

🌊 STREAMING SCENARIOS

🟧 How do you design a watermark strategy for late-arriving Kafka events?

Use `withWatermark(event_time, "delay threshold")`:

```
>> df.withWatermark("event_time", "10 minutes")
```

- Defines how long to **wait for late data**.
- Late events **older than watermark** are **dropped** during aggregation.

Design Tips:

- Choose threshold based on:
    - Expected event delays.
    - Tolerance for late/missing data.
    - State retention limits.

🟧 What's the difference between `trigger(once)` and `trigger(processingTime)`?

`trigger(once)`:

- Runs **once**, processes all available data, then stops.
- Good for **micro-batch jobs that mimic batch processing**.

`trigger(processingTime="5 seconds")`:

- Runs **every interval**, checking for new data.
- Keeps streaming jobs alive

🟧 **How do you recover from a checkpoint corruption in structured streaming?**

1. Stop the streaming job.
2. Backup corrupted checkpoint (for inspection).
3. Remove the checkpoint directory.
4. Restart the job with:
    - Clean checkpoint.
    - Optionally use:
      `>> .option("startingOffsets", "earliest")  # Kafka`
5. Consider using **Delta Lake for idempotent sink** to prevent duplicates during reprocessing.