How to Debug Slow Spark Jobs?

Slow Spark jobs can burn through compute resources and frustrate engineers. Debugging performance issues requires a structured approach using **profiling** (**Spark UI**) and **logs** to pinpoint bottlenecks.

1. Understanding the Symptoms

Before diving into logs or tuning parameters, start by identifying performance symptoms:

a. Long job execution time (Jobs taking hours instead of minutes)

b. Straggler tasks (Some tasks significantly slower than others)

c. High shuffle read/write (Spilling data to disk, causing I/O overhead)

d. Memory consumption issues (Out-of-memory errors or excessive garbage collection)

e. Driver bottlenecks (High CPU usage on the driver node).

Case Study: EMR to Databricks Migration — Job Running 4x Slower

We were helping a retail client **migrate their ETL workloads from AWS EMR to Databricks**. The job, which processed **1 billion transactions daily**, ran in **30 minutes on EMR** but **took 2+ hours on Databricks**.

The immediate question: **Why was the job suddenly 4x slower?**

2. Identifying the Bottlenecks

To debug Spark performance, break it down into:

A. Task Execution Time

If **some tasks take much longer**, it could indicate **data skew**.

If **all tasks are slow**, look at shuffle performance or inefficient transformations.

B. Shuffle and Data Skew

Large shuffle writes can lead to **disk I/O bottlenecks**.

Uneven shuffle distribution can create **straggler tasks**.

C. Memory and Garbage Collection

Frequent task failures → Check for **executor OOM errors**.

Excessive GC time (>10% of job duration) → Memory-intensive operations like large joins.

D. Driver Bottlenecks

If the driver CPU is high, check for **collect() or large dataset broadcasting**.

3. Using Spark UI for Profiling in Databricks

Databricks provides deep insights into job execution through the Spark UI. Here's how we used it for debugging:

A. Checking Stage Execution Timeline

Databricks UI → Jobs → Completed Jobs → Stages

1. We found **one stage taking 70% of the total job time**.
2. The **task distribution was uneven**, with some tasks finishing in 2 minutes while others took 15 minutes.

Root Cause: Severe data skew in a key column used for joining fact and dimension tables.

B. Analyzing DAG Visualization

The **DAG (Directed Acyclic Graph) Visualization** in Databricks UI revealed:

Multiple wide transformations (shuffle-heavy operations). **A large join operation with a skewed key** (causing straggler tasks).

Fix: We applied **salting to balance partitions** and used **broadcast joins** to reduce shuffle.

```
Copyfrom pyspark.sql.functions import monotonically_increasing_id


df = df.withColumn("salt", monotonically_increasing_id() % 10)

df_joined = df1.join(df2, ["common_column", "salt"])
```

C. Monitoring Executors and Storage

We checked the **"Executors" tab in Spark UI** and found:

Executors were frequently restarting due to OOM errors. **Shuffle spill exceeded memory limits**, causing disk writes.

Fix: We optimized shuffle memory allocation:

```
Copyspark.sql.shuffle.partitions=200  # Reduce partitions from 2000 to 200
spark.memory.fraction=0.6        # Allocate more memory for execution
```

4. Digging into Logs for Root Cause Analysis in Databricks

Databricks provides **detailed logs** in the **Driver Logs (stderr/stdout)** and execution history.

A. Checking Executor Logs for OOM Errors

We found **OutOfMemoryErrors** in the executor logs:

```
Copygrep -i "OutOfMemoryError" driver-logs.log
```

Fix: Increased executor memory:

```
Copyspark.executor.memory=6g
spark.executor.memoryOverhead=2g
```

B. Debugging Shuffle Performance Issues

The shuffle logs indicated **excessive spill to disk**:

```
Copygrep -i "shuffle" stderr.log
```

Fix: Reduced shuffle partition size and used **broadcast joins**.

```
Copyfrom pyspark.sql.functions import broadcast
df = df1.join(broadcast(df2), "common_column")
```

C. Identifying Long-Running Tasks

We ran a log analysis script:

```
Copygrep -i "task" executor-logs.log | sort -nk3 | tail -10
```

Fix: Repartitioned based on skewed keys.

```
Copydf = df.repartition(100, "key_column")
```

5. Final Optimizations and Performance Gains

After applying **targeted fixes**, the job execution time **dropped from 2+ hours to 28 minutes**, slightly better than EMR.

Key Improvements:

Balanced partitions to avoid skew Reduced shuffle operations using broadcast joins I**ncreased executor memory to handle large datasets** Op**timized memory management to reduce GC time**

Conclusion

Debugging slow Spark jobs in **Databricks** requires:

1. **Using Spark UI** to identify slow stages and tasks.
2. **Checking logs** for OOM errors, shuffle performance, and executor restarts.
3. **Applying optimizations** like **repartitioning, broadcasting, and memory tuning**.

By following these steps, **80% of performance issues** can be resolved with **minimal tuning**.