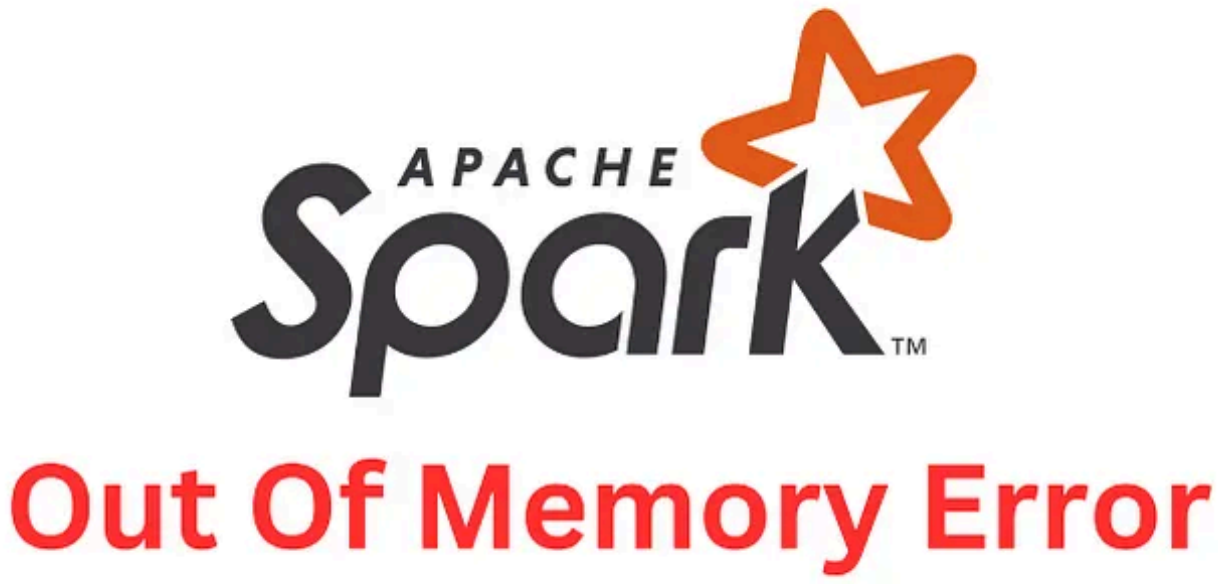


Spark's Out-of-Memory Problems: How Breaking the DAG with S3 Writes Saved Our Jobs

Anurag Sharma 5 min read · Jun 9, 2025

At our Analytics company, we rely heavily on Apache Spark running on AWS EMR clusters to process massive datasets. However, like many teams working with big data, we repeatedly hit a frustrating roadblock: **out-of-memory (OOM) errors**. Over time, we tried multiple optimization techniques, from caching and persisting to using memory-optimized EC2 instances. Each approach worked temporarily, but the OOM issue kept resurfacing.

Finally, we stumbled upon a simple yet effective solution: **breaking the DAG by writing intermediate data to S3 in Parquet format**. This approach not only resolved our OOM issues but also proved more sustainable than our previous attempts.

The Persistent Out-of-Memory Problem

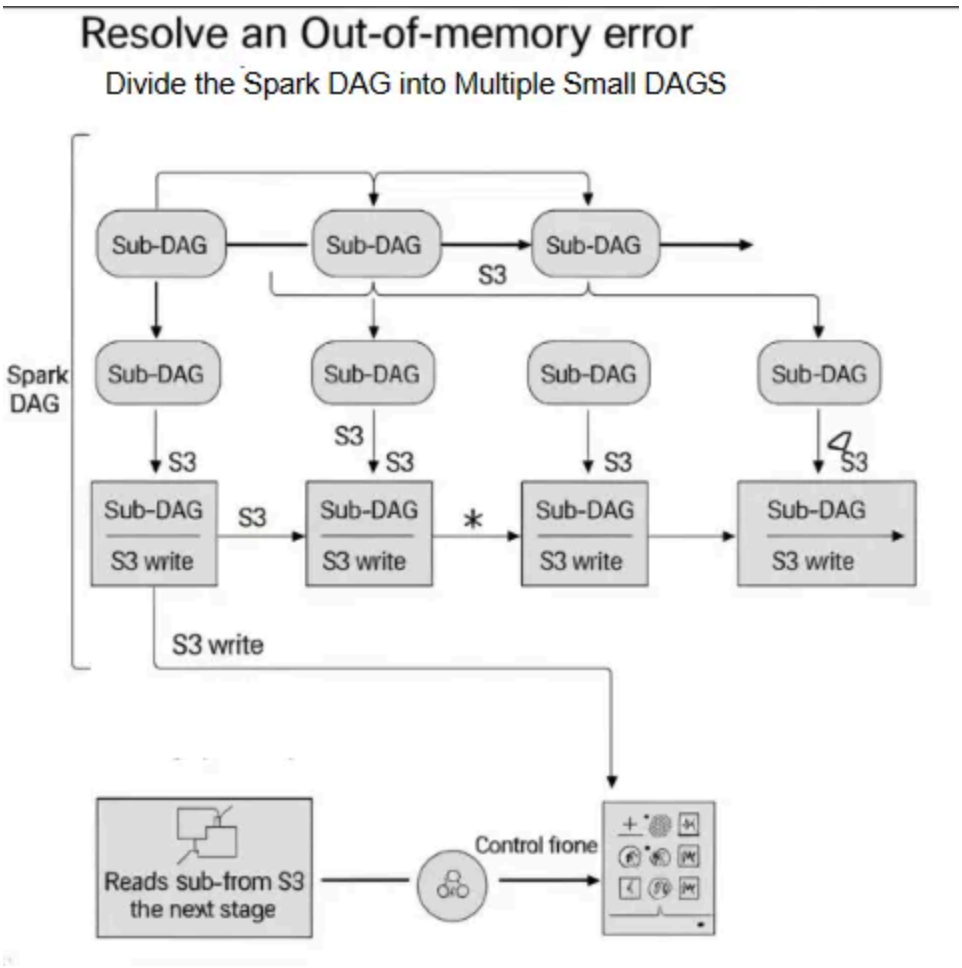
Our Spark jobs, designed to handle complex transformations and joins on large datasets, would periodically fail with OOM errors. These failures disrupted our workflows, delayed insights, and frustrated our team. Initially, we tackled the issue with standard Spark optimization techniques:

1. Caching and Persisting: We used persist() and cache() to store intermediate DataFrames in memory, reducing recomputation. This worked well for about a year, but as data volumes grew, the OOM errors returned.
2. Optimizing Joins and Filters: We refined our queries by filtering data early, removing unnecessary columns, and broadcasting smaller tables to minimize shuffle overhead. These changes kept the job running for another five months before OOM struck again.
3. Memory-Optimized EC2 Instances: We upgraded to memory-optimized EC2 instances in our EMR cluster, expecting a significant boost. While this extended our job's lifespan by three months, the OOM issue persisted.
Despite the expertise of our team, each solution was a temporary fix. The root cause — **data shuffling** during complex transformations and joins — was overwhelming Spark's memory management.

The Breakthrough: Debugging with Spark UI
Frustrated by recurring failures, we decided to take a step back and adopt a simpler, more methodical approach. Using the **Spark UI**, we dove deep into

the job's execution plan to pinpoint the exact source of the OOM errors. The culprit was clear: **excessive shuffling**. Shuffling, a necessary evil in distributed computing, involves redistributing data across nodes, often leading to memory spikes and performance bottlenecks.

While this wasn't a new revelation, our team hadn't fully addressed the impact of shuffling on our DAG (Directed Acyclic Graph). Spark's DAG represents the sequence of transformations and actions in a job, and long, complex DAGs can exacerbate memory issues during shuffles. This insight led us to a novel idea: **break the DAG by writing intermediate results to S3 in Parquet format.**



The Solution: Breaking the DAG with S3 Writes

Instead of letting Spark maintain a long, memory-intensive DAG, we

introduced a deliberate break after each shuffle-heavy operation.

Here is how we did it:

1. Write Intermediate Results to S3: After every significant shuffle (e.g., joins or aggregations), we wrote the intermediate DataFrame to S3 in Parquet format.

2. Read Back from S3: We then read the intermediate data from S3 to continue the next stage of processing.

This approach effectively "reset" the DAG at each write, preventing Spark from holding excessive lineage information in memory. By breaking the DAG, we reduced the memory footprint of the job, as Spark no longer needed to track the entire computation graph across multiple shuffles.

Why This Worked Better Than Previous Approaches

While writing to S3 increased job runtime by 10–15% (due to I/O overhead), the trade-off was well worth it.

Here is why this approach outperformed our earlier attempts:

1. Reduced Memory Pressure:

◦ **Previous Approaches:** Caching/persisting kept data in memory, which worked until data sizes exceeded available RAM. Optimizing joins and filters helped, but shuffles still caused memory spikes. Memory-optimized instances provided more headroom but didn't address the root issue of shuffle-related memory demands.

◦ **S3 Writes:** Writing intermediate results to S3 offloads data from memory to

disk, freeing up resources during execution. By breaking the DAG, we prevented Spark from holding onto large lineage graphs, which reduced memory usage significantly.

2. Improved Fault Tolerance:
◦ **Previous Approaches**: If a job failed mid-execution, Spark had to recompute the entire DAG from the source, which was time-consuming and resource-intensive.
◦ **S3 Writes**: Writing intermediate results to S3 created checkpoints. If a job failed, we could restart from the last checkpoint, avoiding costly recomputations.

3. Scalability:
◦ **Previous Approaches**: Optimizations like broadcasting or using larger instances worked temporarily but didn't scale as data volumes grew.
◦ **S3 Writes**: Writing to S3 is inherently scalable, as S3 can handle massive datasets without memory constraints. This made our solution robust against future data growth.

4. Simplicity and Maintainability:
◦ **Previous Approaches**: Complex optimizations (e.g., fine-tuning joins or managing cache levels) required deep expertise and constant tweaking, making maintenance difficult.
◦ **S3 Writes**: The approach was straightforward — write to S3 after shuffles and read back. This simplicity made it easier for our team to implement and maintain.

5. Long-Term Stability:
◦ Unlike our previous attempts, which lasted 3–12 months before failing, the S3-based approach kept our jobs running without OOM errors for a

significantly longer period. The slight increase in runtime was a small price to pay for reliability.

Lessons Learned: Simplicity Wins

Our journey taught us a valuable lesson: **sometimes, simpler is better**. While advanced optimizations and fancier hardware seemed appealing, they only delayed the inevitable.

By leveraging Spark UI to identify shuffling as the bottleneck and adopting a straightforward solution — writing to S3 in Parquet — we achieved a robust, scalable, and maintainable fix.

Key Takeaways for Spark Users

If you're battling OOM errors in Spark, consider these steps:

• **Use Spark UI**: Dive into the UI to understand your job's execution plan and identify shuffle-heavy stages.

• **Break the DAG**: Write intermediate results to a durable storage layer like S3 in Parquet format to reduce memory pressure and improve fault tolerance.

• **Accept Trade-Offs**: A slight increase in runtime can be a worthwhile trade-off for stability and scalability.

• **Keep It Simple**: Complex optimizations may provide short-term gains, but simpler solutions often yield long-term success.

By breaking the DAG with S3 writes, we transformed a recurring nightmare

into a reliable process. This approach not only solved our OOM issues but also gave us confidence in handling growing datasets. If you're facing similar challenges, give this method a try — it might just be the game-changer you need.

Data Engineering | Data Engineer | Optimization | Spark | Data Analytics

Written by Anurag Sharma

82 followers · 3 following

Edit profile

Data Engineering Specialist with 10+ exp. Passionate about optimizing pipelines, data lineage, and Spark performance and sharing insights to empower data pros!

Responses (1)

Anurag Sharma him/he

What are your thoughts?

Veeramani
Aug 12, 2025

What a detailed explanation with solution for OOM. Thank you so much to write this @Anurag Sharma