



From Out-of-Memory to Optimized: Handling Java Heap Space & GC Overhead Limit Exceeded issues in Databricks



Mohit Joshi

Follow

8 min read · Sep 26, 2024



29



1



In large-scale data processing using Apache Spark, memory-related issues like “Java Heap Space Out of Memory” and “GC Overhead Limit Exceeded” are common, especially in environments like Databricks. These errors can significantly affect the performance and reliability of Spark jobs, leading to failed tasks and prolonged runtimes. This article dives into these errors, exploring their causes, the specific Spark configuration challenges in Databricks, and the solutions we implemented to overcome them. By understanding our scenario and optimizations, readers can gain insights into resolving similar issues in their own Spark workloads.

Part One: Tackling Java Heap Space Errors in Databricks

Medium

Sign up to discover human stories that deepen your understanding of the world.

Free

- ✓ Distraction-free reading. No ads.
- ✓ Organize your knowledge with lists and highlights.
- ✓ Tell your story. Find your audience.

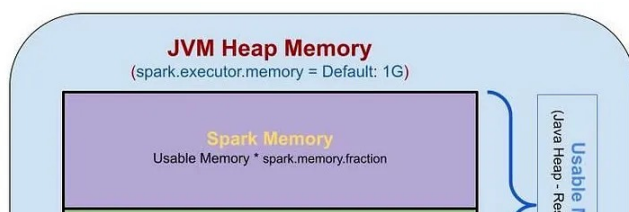
Sign up for free

Membership

- ✓ Read member-only stories
- ✓ Support writers you read most
- ✓ Earn money for your writing
- ✓ Listen to audio narrations
- ✓ Read offline with the Medium app

Try for \$5/month

- **Spark Memory:** Divided into two parts:
 - **Execution Memory:** Handles computation tasks such as shuffling, sorting, joins, and aggregations.
 - **Storage Memory:** Stores cached and persisted data like RDDs and DataFrames.
- **Reserved Memory:** A small portion reserved for internal Spark operations to prevent out-of-memory errors during critical tasks.
- **User Memory:** Memory allocated to user-defined objects and data structures within your Spark application.



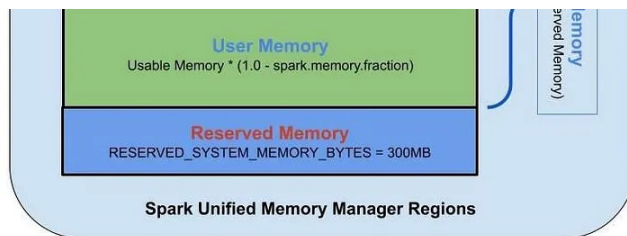


Fig. Java Heap Memory Distribution for Spark

On-Heap vs. Off-Heap Memory:

- **On-Heap Memory:** This is the default memory management method where data is stored directly in the Java Heap, making it subject to garbage collection (GC). Inefficient GC can lead to errors like “GC Overhead Limit Exceeded.”
- **Off-Heap Memory:** Managed outside the JVM, off-heap memory reduces GC overhead and can improve performance, especially with large datasets.

1.2 Understanding the “java.lang.OutOfMemoryError: Java heap space”

The java.lang.OutOfMemoryError: Java heap space error happens when Spark applications running on Databricks run out of memory allocated to the Java Heap. This is typically caused by the JVM being unable to allocate enough memory for processing objects. In Databricks, the error is usually triggered by:

- **Excessive Data Loading:** Large datasets are loaded into memory without filtering or partitioning.
- **Inefficient Caching:** Over-caching or improper storage levels consume more memory than allocated.
- **Skewed Data:** Some partitions hold disproportionately large amounts of data, putting pressure on memory.
- **Large Shuffle Operations:** Wide transformations like joins, groupBy, or aggregations can overwhelm the system during shuffling operations.

1.3 Databricks Memory Calculation: Demystified

Let's say you have the following Databricks cluster setup:

- **Driver Node:** 16GB RAM, 4 Cores
- **Worker Nodes:** 16GB RAM, 4 Cores, with auto-scaling from 2 to 8 worker nodes

By default, Databricks assigns one executor per worker node. To optimize performance, you must understand how memory is allocated in Databricks.

Default Spark Memory Configurations:

- `spark.memory.fraction` : 0.75
- `spark.memory.storageFraction` : 0.5

Using the above defaults, the memory breakdown for each executor looks like this:

- **Total Heap Space per Executor:** 16GB
- **Reserved Memory:** 300MB
- **User Memory:** $(1 - 0.75) * 16GB = 4GB$
- **Spark Memory:** $16GB - 4GB - 300MB \approx 12GB$
- **Storage Memory:** $0.5 * 12GB = 6GB$
- **Execution Memory:** $0.5 * 12GB = 6GB$

However, when checking the Spark UI, I found that on-heap storage memory was only 4.8GB. Upon further investigation, I realized Databricks uses a

custom formula to calculate Spark executor memory: $(\text{all_memory_size} * 0.97 - 4800\text{MB}) * 0.8$

- **All Memory Size:** Total RAM of the executor
- **0.97:** Accounts for kernel overhead
- **4800MB:** Reserved for internal services like node daemons and log services
- **0.8:** is a heuristic to ensure the LXC container running the Spark process doesn't crash due to out-of-memory errors.

Using this formula, the actual memory for each executor is:

$(16\text{GB} * 0.97 - 4800\text{MB}) * 0.8 \approx 8.5\text{GB}$

This explains why the on-heap storage memory was 4.8GB ($8.5\text{GB} * 0.6$).

Based on these calculations, we adjusted the cluster size to better accommodate our needs.

When we found out about databricks memory calculations then we increased the cluster size as well.

1.4 Resolving Java Heap Space Errors

Now that we understand the memory allocation and error causes, let's discuss how to prevent and fix Java Heap Space errors in Databricks.

1.4.1 Optimize Memory and Cluster Configuration

- Calculate your memory requirements based on caching and transformations.
- Tune `spark.memory.fraction` and `spark.memory.storageFraction` based on your workload. For example:
 - If you're caching more data than shuffling, increase `spark.memory.storageFraction`.
 - If your application involves more shuffling, decrease the storage fraction to allocate more memory to execution.

1.4.2 Use Caching and Unpersisting Wisely

Cache DataFrames only when an action is called multiple times on the same data to avoid re-running the DAG. Don't forget to unpersist the data once it's no longer needed to free up memory.

```
df = spark.read.csv('file_path.csv')
# Apply transformations
df = df.cache()
count = df.count() # Action triggers caching
df.write.csv('file_path.csv')
df.unpersist() # Free memory after usage
```

Over-caching or neglecting to unpersist can fill up Spark storage memory, leading to heap space errors.

1.4.3 Avoid Overusing collect()

The `df.collect()` method can be very memory-intensive because it brings all distributed data back to the driver, which has limited memory. For example:

```
count = len(list(df.select(df.columns[0]).rdd.flatMap(lambda x: x).collect()))
```

Though faster than `df.count()`, this method caused memory issues in our case due to excessive data collection on the driver. We replaced it with more efficient alternatives to avoid out-of-memory errors.

1.4.4 Leverage AQE for Efficient Shuffling

Adaptive Query Execution (AQE) dynamically optimizes Spark queries based on real-time data statistics. It can:

- Adjust join strategies (e.g., switch from sort-merge to broadcast joins)
- Combine small partitions after shuffle.
- Handle skewed data by splitting large partitions

Using AQE can reduce the chances of memory errors during shuffles.

```
spark.databricks.optimizer.adaptive.enabled=true
```

By applying these optimizations, you can significantly reduce the likelihood of encountering Java Heap Space errors in Databricks and improve the performance of your Spark applications.

Part Two: Resolving the “GC Overhead Limit Exceeded” Error in Databricks

Encountering the “GC Overhead Limit Exceeded” error alongside a Java Heap Space issue is common when working with large-scale Spark applications on Databricks. Let’s explore what this error is, why it occurs, and how to mitigate it.

2.1 Understanding Garbage Collection in Spark

Garbage Collection (GC) is a critical process in Java-based applications, including **Apache Spark**. It is the mechanism by which the Java Virtual Machine (JVM) automatically reclaims memory that is no longer in use, helping prevent memory leaks and ensuring your application runs smoothly.

As Spark applications process massive datasets, they create numerous temporary objects in memory. If these objects aren’t disposed of properly, they accumulate, leading to **memory exhaustion** and, potentially, application crashes. GC helps by periodically freeing up memory, but if not well-optimized, it can lead to performance issues like the “GC Overhead Limit Exceeded” error.

2.2 What Is the “GC Overhead Limit Exceeded” Error?

The “GC Overhead Limit Exceeded” error occurs when the JVM spends more than **98% of its time** on garbage collection but is able to free less than 2% of memory. This suggests that the JVM is stuck in an endless loop of GC without successfully reclaiming enough memory, often due to inefficient memory usage or configuration problems.

In Spark applications, this issue arises under conditions of **heavy memory pressure**, such as when large datasets are processed without proper optimization. The JVM keeps trying to reclaim space in the heap, but since there is little available memory, it ends up triggering the error.

Common Causes of the “GC Overhead Limit Exceeded” Error:

- **Insufficient Memory Allocation:** The executor has too little memory allocated to handle the workload.
- **Data Skew:** Imbalanced data distribution across partitions results in some executors working harder, leading to memory pressure.
- **Inefficient Algorithms:** Algorithms that create excessive intermediate data or perform unnecessary computations increase memory usage.
- **Excessive Task Scheduling:** Too many tasks can lead to memory fragmentation, increasing GC cycles.
- **Overuse of Caching and Unpersisting:** Frequent and excessive caching without proper unpersisting puts pressure on memory, leading to frequent GC operations.

2.3 Solutions to the “GC Overhead Limit Exceeded” Error

Let’s go one by one and explore some solutions for this error.

2.3.1 Reduce Memory Pressure

Reducing memory pressure can significantly alleviate GC overhead by lowering the frequency of garbage collection cycles. Here are some strategies to reduce memory pressure in Spark:

- **Increase Memory Allocation:** Increase the available memory by adjusting the `spark.memory.fraction` parameter, as explained in the Java Heap Space section.
- **Handle Skewed Data:** Enable **Adaptive Query Execution (AQE)** to handle skewed data efficiently, preventing memory overload on individual nodes.
- **Optimize Task Scheduling:** Avoid unnecessary transformations and shuffle operations. Fine-tune the `spark.sql.shuffle.partitions` setting to align with your data size and the number of cores available.
- **Use Caching Carefully:** Limit the use of caching, and ensure you unpersist datasets after use to avoid unnecessary memory consumption.

2.3.2 Switch to a More Efficient Garbage Collection Algorithm

A more efficient garbage collection algorithm can improve memory management and reduce GC overhead for Spark applications, especially when dealing with large datasets. The **default GC algorithm** may not always be optimal for high-performance, large-scale Spark jobs.

Consider switching to **G1GC (Garbage First GC)** or **CMS (Concurrent Mark-Sweep GC)**, which are better suited for large heaps and can reduce GC pauses. These algorithms allow for more efficient memory management and faster garbage collection cycles.

To change the garbage collection algorithm in Spark, you can modify the configuration with:

```
spark.executor.extraJavaOptions -XX:+UseG1GC
```

Alternatively, for **Concurrent Mark-Sweep GC**:

```
spark.executor.extraJavaOptions -XX:+UseConcMarkSweepGC
```

By implementing the appropriate GC algorithm, you can enhance memory efficiency and avoid excessive garbage collection, ultimately reducing the chances of encountering the **GC Overhead Limit Exceeded** error.

Conclusion: Key Strategies for Overcoming Java Heap and GC Errors in Databricks

In summary, handling memory issues like **Java Heap Space errors** and **GC Overhead Limit Exceeded** errors in Databricks requires a combination of understanding, optimization, and configuration:

1. **Tuning Spark Configurations:** Carefully adjust spark configurations like `spark.executor.memory`, `spark.memory.fraction`, and `spark.memory.storageFraction` to fit your workload's needs.
2. **Use Caching and Unpersisting Wisely:** Cache data only when necessary and always unpersist unused data to avoid excessive memory consumption.
3. **Leverage Adaptive Query Execution (AQE):** Enable AQE to efficiently handle skewed data and optimize shuffling, reducing memory strain.
4. **Switch to Efficient GC Algorithms:** Consider using G1GC or CMS for improved garbage collection, especially for large-scale Spark applications.
5. **Monitor Regularly:** Regularly monitor Spark UI and application logs to identify memory bottlenecks.

Article References

1. **Decoding Memory in Spark: Parameters That Are Often Confused**
[Walmart Global Tech — Medium](#)

2. Apache Spark Memory Management
[Analytics Vidhya — Medium](#)
3. Why Spark UI Shows Less Memory Than Expected in Databricks
[Databricks Knowledge Base](#)
4. Configure Clusters for Databricks
[Databricks Documentation](#)
5. Tuning Spark Applications
[Apache Spark Official Documentation](#)
6. How to Change the Number of Executors/Instances in Databricks
[Databricks Community Forum](#)
7. Adaptive Query Execution (AQE) in Databricks
[Databricks Documentation](#)
8. Garbage Collection in Spark: Why It Matters and How to Optimize It
[Siraj Deen — Medium](#)

Spark

Spark Optimization

Databricks

Data Engineering

Databricks Workflows



29



1



Published in Dev Genius

27K Followers · Last published 21 hours ago

Coding, Tutorials, News, UX, UI and much more related to development

Follow



Written by Mohit Joshi

10 Followers · 3 Following

I'm Data + Software Engineer. Passionate reader & learner of new technologies & enthusiastic to share simplified tech knowledge.

Follow

Responses (1)



Write a response

What are your thoughts?



Stuti Mishra

Sep 26, 2024



You've explained a very relevant and common issue in a clear and straightforward way.



Reply

More from Mohit Joshi and Dev Genius

In Dev Genius by Mohit Joshi

Mastering Spark DAGs: The Ultimate Guide to Understanding...

If you've ever worked with Apache Spark, you've probably heard about DAGs (Directe...

Feb 7



7



In Dev Genius by Anil R

Spring Boot Red Flags: What You Should Never Do

Spring Boot is a powerful framework designed to simplify Java application...



Mar 31




216




10




 In Dev Genius by Anusha SP

Java 8 Coding and Programming Interview Questions and Answers

It has been 8 years since Java 8 was released. I have already shared the Java 8 Interview...

Jan 31, 2023  1.4K  21



 In Dev Genius by Mohit Joshi

Demystifying Spark Data Lineage: How to Track and Rebuild Data...

Imagine you're working with massive amounts of data. As the data moves through...


Feb 19  5



See all from Mohit Joshi

See all from Dev Genius

Recommended from Medium


 Shantanu Tripathi

Sort Merge Join: Visual Flow inside Executors

Learn the default join strategy used by Spark through visuals.

 Dec 29, 2024  60





 In Data Engineer Things by Vu Trinh

I spent 4 hours learning Apache Spark Resource Allocation



Spark's resource allocation mechanism and the two scheduling modes.

 Nov 2, 2024  164  1



 Martin Jurado Pedroza 

Lambda functions and built-in functions (Python and Pyspark)


 Nov 23, 2024  3



 Santosh Joshi

Study Notes: Spark Execution - Jobs, Stages, Tasks, and Slots...

A Quick Guide into Spark's Execution Model for Data Engineers

 Nov 30, 2024  75




 Siddharth Ghosh

Spark Interview Series — Difference between Cache and...

Apache Spark has become one of the most popular big data processing frameworks,...

 Apr 11



 In Dev Genius by Mohit Joshi

Mastering Spark DAGs: The Ultimate Guide to Understanding...

If you've ever worked with Apache Spark, you've probably heard about DAGs (Directe...

Feb 7  7



[See more recommendations](#)

[Help](#) [Status](#) [About](#) [Careers](#) [Press](#) [Blog](#) [Privacy](#) [Rules](#) [Terms](#) [Text to speech](#)