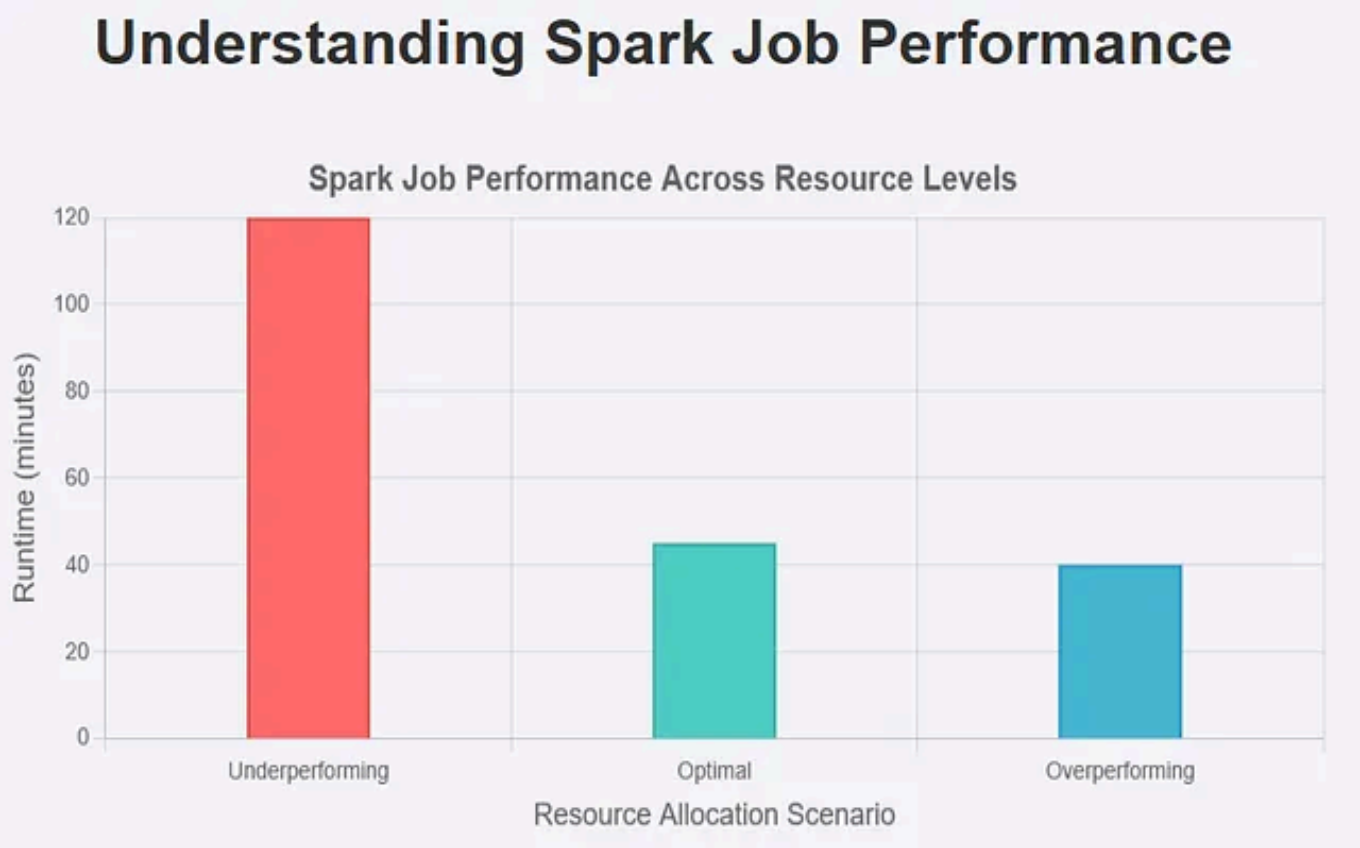


Understanding Spark Job Performance: Is Your Job Underperforming or Overperforming with Resource Allocation?

Anurag Sharma 7 min read · Jul 21, 2025

Apache Spark is a powerful distributed computing framework, but its performance heavily depends on proper resource allocation. Misconfigured resources can lead to underperforming jobs that run slowly or overperforming jobs that waste resources. In this blog, we'll explore how to evaluate whether your Spark job is underperforming or overperforming in terms of resource allocation, and we'll provide practical tips to optimize your Spark jobs for efficiency and speed.

What Does Underperforming or Overperforming Mean?

Before diving into the details, let's define what we mean by underperforming and overperforming in the context of Spark resource allocation:

• **Underperforming**: Your Spark job is running slower than expected, failing to meet performance SLAs, or encountering failures (e.g., out-of-memory errors) due to insufficient resources like CPU, memory, or disk space.

• **Overperforming**: Your Spark job is completing successfully but is consuming more resources than necessary, leading to inefficiencies, higher costs, or reduced cluster availability for other jobs.

Both scenarios can negatively impact your data pipeline, either by delaying results or inflating infrastructure costs. Let's explore how to identify these issues and optimize resource allocation.

Key Metrics to Monitor Spark Job Performance:

To determine whether your Spark job is underperforming or overperforming, you need to monitor key metrics and analyze resource usage.

Here are the primary areas to focus on:

1. Job Execution Time

• **What to Look For:** Compare the job's execution time to historical runs or expected SLAs. A job taking significantly longer than expected may indicate underperformance due to insufficient resources. Conversely, a job completing much faster than necessary with excessive resources may suggest overperformance.

• **How to Check:** Use the Spark UI to track job duration, stage times, and task execution times.

2. Resource Utilization

• **CPU Usage:** Check whether executors are fully utilizing allocated CPU cores. Low CPU usage (<50%) across executors may indicate over-allocation, while consistently maxed-out CPU usage (100%) suggests under-allocation.

• **Memory Usage:** Monitor memory consumption in the Spark UI's "Executors" tab. High memory usage with frequent garbage collection or out-of-memory errors points to underperformance. Conversely, low memory usage (0%) may indicate over-allocation.

• **How to Check:** Use tools like Spark UI, Ganglia, or cluster monitoring tools (e.g., Datadog, Prometheus) to track CPU and memory metrics.

3. Data Skew and Task Imbalance

• **What to Look For:** Uneven task execution times or data skew can cause some executors to be overloaded while others remain idle. This often leads to underperformance, as resources are not utilized efficiently.

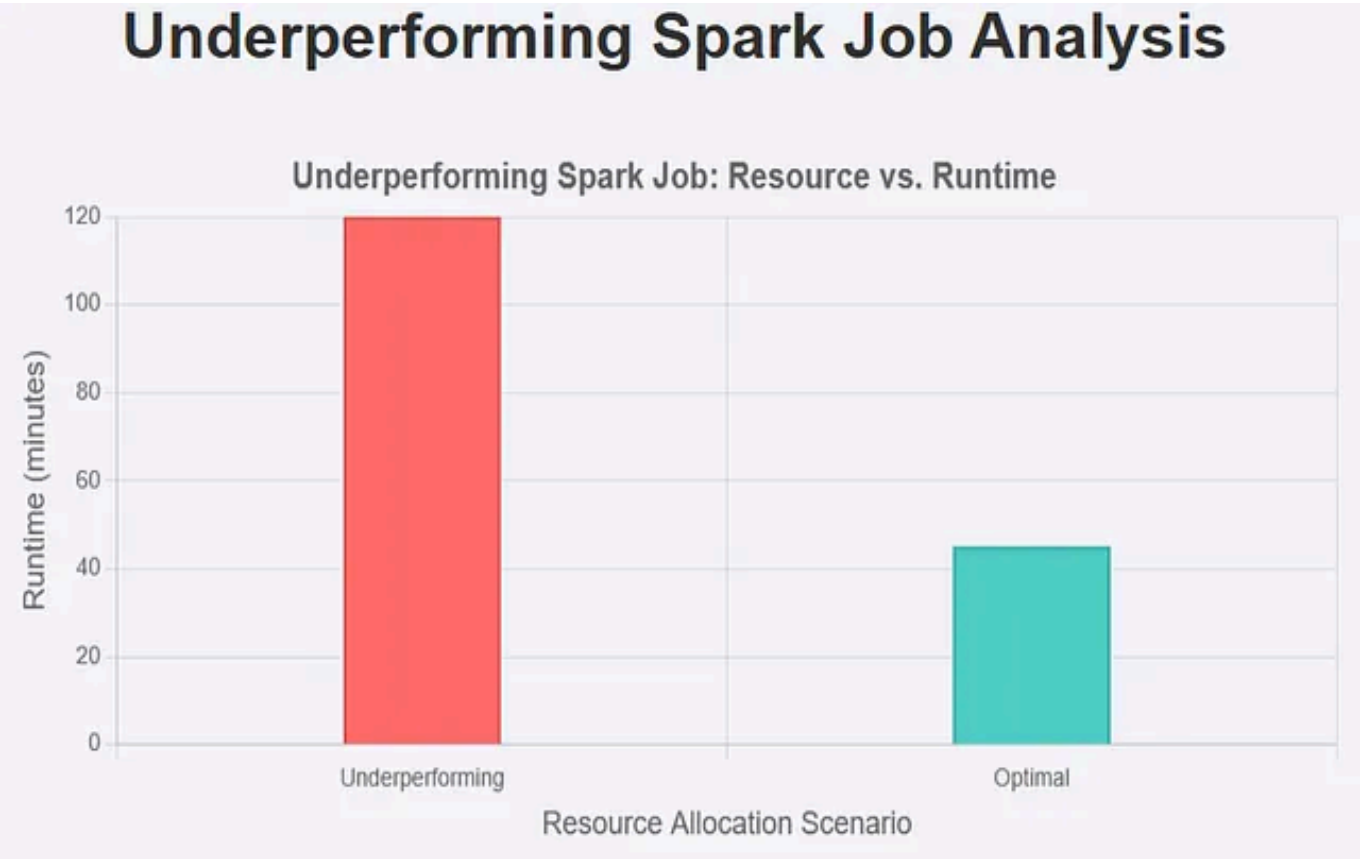
• **How to Check:** In the Spark UI, look at the "Tasks" section for stages with significant variance in task execution times. Check the "Summary Metrics" to identify tasks with disproportionately large input data sizes.

4. Disk and Network I/O

• **What to Look For:** Excessive disk spills (data written to disk due to insufficient memory) or high network I/O (e.g., during shuffles) can indicate underperformance. Overprovisioned resources may result in minimal disk or network usage, suggesting inefficiency.
• **How to Check:** The Spark UI's "Stages" tab shows metrics like "Shuffle Read/Write" and "Disk Spill." Cluster monitoring tools can also track I/O metrics.

5. Cluster Resource Allocation

• **What to Look For:** Compare allocated resources (cores, memory, executors) to actual usage. Overprovisioning leads to idle resources, while underprovisioning causes bottlenecks.
• **How to Check:** Review the Spark configuration settings (e.g., spark.executor.cores, spark.executor.memory) and compare them to utilization metrics in the Spark UI or cluster manager (e.g., YARN, Kubernetes).



Signs Your Spark Job Is Underperforming

Here are common indicators that your Spark job is underperforming due to insufficient resource allocation:

1. Long Execution Times: Tasks take too long to complete, or the job fails to meet performance SLAs.

2. Out-of-Memory Errors: Executors crash with OOM errors, often visible in logs or the Spark UI.

3. Frequent Garbage Collection: Excessive garbage collection pauses indicate insufficient memory for the workload.

4. High Disk Spills: Data spills to disk because the allocated memory is insufficient, slowing down processing.

5. Data Skew: Some tasks process significantly more data than others, causing bottlenecks and uneven resource usage.

6. High Shuffle I/O: Large shuffles with insufficient memory or network bandwidth lead to slow performance.

Underperforming Example Scenario:

Suppose you're running a Spark job that processes 1 TB of data but consistently fails with OOM errors. The Spark UI shows high disk spills and frequent garbage collection. This suggests that the job is underperforming due to insufficient executor memory or too few executors.

Signs Your Spark Job Is Overperforming

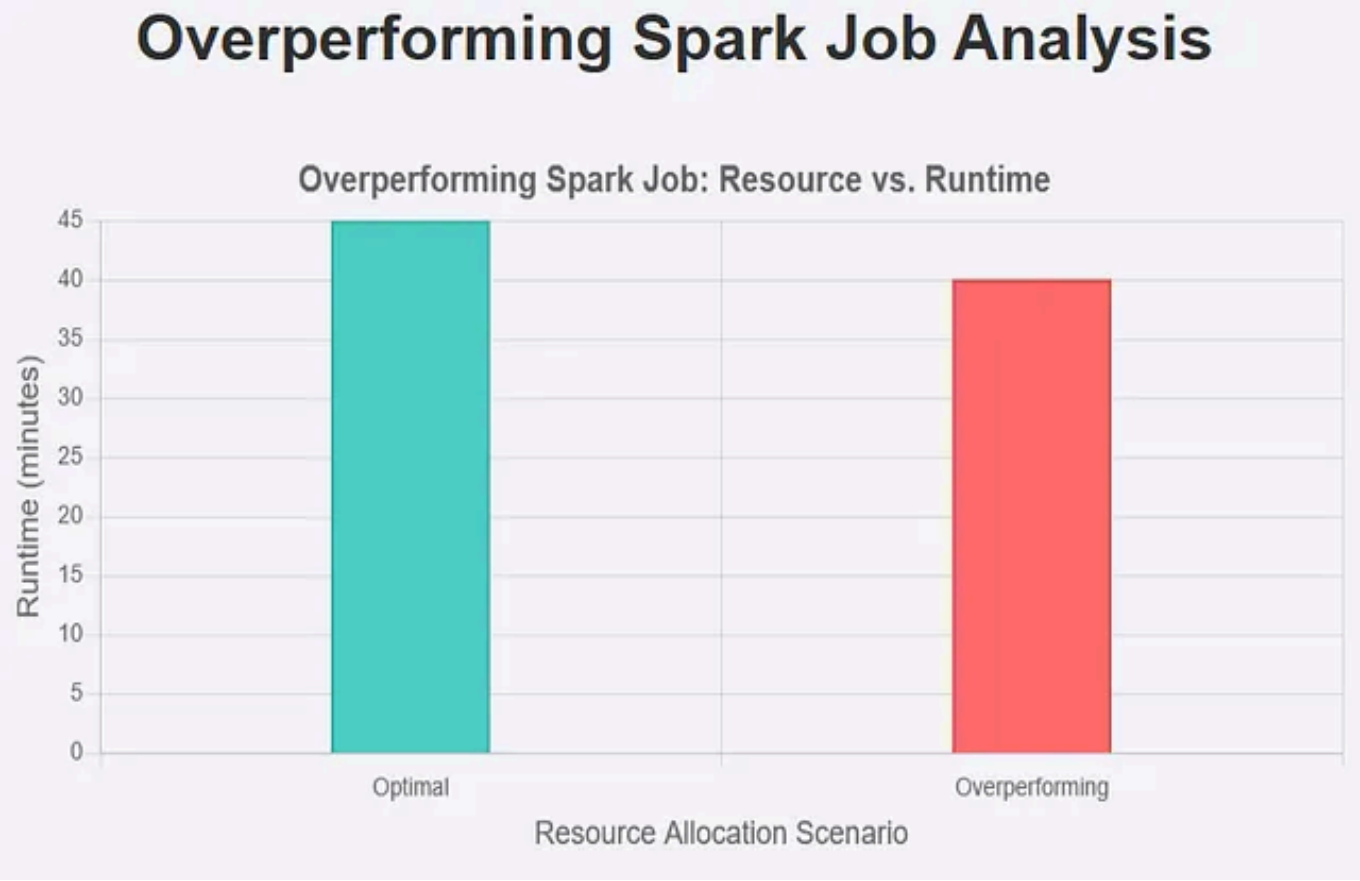
Overperforming jobs waste resources, increasing costs and reducing cluster availability. Here are signs of overperformance:

1. Low Resource Utilization: Executors use only a fraction of allocated CPU or memory (e.g., 0% utilization).

2. Excessive Executors: The job uses more executors than necessary, with many idle or underutilized.

3. Minimal Disk or Network Usage: No disk spills or low shuffle activity may indicate over-allocated memory or executors.

4. Faster-Than-Needed Execution: The job completes much faster than required, suggesting resources could be scaled down without impacting performance.



Overperforming Example Scenario:
Imagine a Spark job processing 100 GB of data completes in 5 minutes, but the SLA allows 15 minutes. The Spark UI shows CPU usage at 20% and memory usage at 10%. This indicates overperformance, as the job is over-allocated resources that could be reduced to save costs.

How to Optimize Resource Allocation
Once you've identified whether your Spark job is underperforming or overperforming, you can take steps to optimize resource allocation.

Here's a step-by-step guide:
1. Analyze Spark UI and Logs
• Use the Spark UI to identify bottlenecks like data skew, high shuffle I/O, or excessive garbage collection.

• Check logs for errors (e.g., OOM, task failures) and warnings about resource contention.

2. Tune Executor Resources

• **For Underperformance**:

◦ Increase spark.executor.memory to reduce disk spills and OOM errors.

◦ Add more executors (spark.executor.instances) or cores (spark.executor.cores) to parallelize tasks.

◦ Adjust spark.memory.fraction and spark.memory.storageFraction to optimize memory allocation between execution and storage.

• **For Overperformance**:

◦ Reduce spark.executor.memory or spark.executor.cores to match actual usage.

◦ Decrease the number of executors to free up cluster resources.

3. Address Data Skew

• Use techniques like salting keys, repartitioning, or custom partitioning to distribute data evenly across tasks.

• Increase spark.sql.shuffle.partitions to reduce the size of individual tasks during shuffles.

4. Optimize Shuffles

• Minimize shuffles by using operations like reduceByKey instead of groupByKey.

• Increase spark.shuffle.memoryFraction or use faster storage (e.g., SSDs) to reduce shuffle I/O bottlenecks.

5. Experiment with Dynamic Allocation

• Enable spark.dynamicAllocation.enabled to allow Spark to adjust the number of executors based on workload, reducing overprovisioning.

• Set appropriate min/max executor limits (spark.dynamicAllocation.minExecutors, spark.dynamicAllocation.maxExecutors).

6. Monitor and Iterate

• Continuously monitor resource utilization and job performance after making changes.

• Use tools like Spark UI, Prometheus, or Datadog to track metrics over time and identify trends.

> ***Practical Example: Diagnosing and Fixing a Spark Job***
> *Let's walk through an example of diagnosing and optimizing a Spark job.*
> ***Scenario***
> *A Spark job processes a 500 GB dataset but takes 2 hours to complete, exceeding the 1-hour SLA. The Spark UI shows:*
> *• High disk spills (50 GB spilled to disk).*
> *• Frequent garbage collection pauses.*
> *• CPU usage is at 90% across all executors.*
> *• Significant data skew in one stage, with some tasks processing 10x more data than others.*
> ***Diagnosis***
> *• **Underperformance**: The job is underperforming due to insufficient memory (causing disk spills and garbage collection) and data skew.*
> *• **Resource Bottlenecks**: High CPU usage suggests the job needs more parallelism or better data distribution.*

Optimization Steps

1. Increase Memory: Increase spark.executor.memory from 4 GB to 8 GB to reduce disk spills.

2. Add Executors: Increase spark.executor.instances from 10 to 20 to

improve parallelism.
3. Fix Data Skew: Repartition the data using a custom key or increase spark.sql.shuffle.partitions to 1000.
4. Monitor Results: Rerun the job and check the Spark UI to confirm reduced disk spills and balanced task execution.
After optimization, the job completes in 50 minutes, meeting the SLA with improved resource efficiency.

Tools to Aid Performance Analysis
To make diagnosing and optimizing Spark jobs easier, leverage these tools:
• **Spark UI**: Provides detailed metrics on job execution, resource usage, and bottlenecks.
• **Cluster Managers**: YARN, Kubernetes, or Mesos dashboards show cluster-wide resource utilization.
• **Monitoring Tools**: Prometheus, Grafana, or Datadog offer advanced visualization of CPU, memory, and I/O metrics.
• **Logging**: Enable verbose logging (log4j.logger.org.apache.spark=DEBUG) to capture detailed diagnostic information.

Conclusion:
Understanding whether your Spark job is underperforming or overperforming requires careful monitoring of execution time, resource utilization, and data distribution. By analyzing metrics in the Spark UI and other monitoring tools, you can identify bottlenecks like insufficient memory, data skew, or over-allocated resources. Optimizing resource allocation through configuration tuning, dynamic allocation, and data skew mitigation can significantly improve job performance and cluster efficiency. Regularly monitor your Spark jobs, experiment with configurations, and iterate based on observed metrics. With these practices, you'll ensure your

Spark jobs run efficiently, meet performance goals, and make the most of your cluster resources.

Spark | Data Analytics | Optimization | Big Data | Databricks

Written by Anurag Sharma

Edit profile

83 followers · 3 following

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

No responses yet

Anurag Sharma him/he

What are your thoughts?

More from Anurag Sharma