Open in app ↗

Medium Search Write 3

Get unlimited access to the best of Medium for less than $1/week. Become a member

Optimizing Job Performance: Redistributing Partitioned Tables into Balanced Buckets

Anurag Sharma 6 min read · Jun 19, 2025

1

In the fast-moving world of data engineering, where data volumes grow very fast and pipelines are the lifeblood of analytics, keeping jobs running smoothly is no small feat. At our company, we faced a growing challenge: our data was expanding at a breakneck pace, causing our jobs to take longer and longer to complete. This led to delayed pipelines, late report refreshes, and frustrated stakeholders. While scaling up our cluster seemed like an obvious fix, it came with a hefty price tag and wasn't always the smartest

solution. Instead, we found a simpler, more effective approach: redistributing our high-volume partitioned tables into the correct number of buckets.

The Problem: Growing Data, Sluggish Jobs, and Costly Fixes

Our data ecosystem was handling terabytes of data, with daily growth that showed no signs of slowing down. We were running hundreds of Spark jobs daily to process this data, joining massive tables like `Customers` and `Orders` to power critical reports and dashboards. However, as our data ballooned, our jobs started to lag:

- **Delayed Pipelines**: Jobs that once took 30 minutes were now stretching into hours, pushing pipeline SLAs past their deadlines.
- **Late Reports**: Downstream reports and dashboards were delayed, impacting business decisions that relied on timely data.
- **Resource Strain**: The longer runtimes consumed more cluster resources, leading to contention and further slowdowns.

Scale up the cluster — add more nodes, increase compute power, and process more data faster. But this approach had serious downsides:

- **Cost Explosion**: Larger clusters meant higher cloud compute bills, eating into our budget.
- **Inefficiency**: Throwing more resources at the problem didn't address the root cause.
- **Diminishing Returns**: Beyond a certain point, adding nodes offered marginal gains due to overheads like network latency and task scheduling.

- **Maintenance Overhead**: Bigger clusters required more monitoring and tuning, adding to our operational burden.

We needed a smarter, more sustainable solution — one that optimized our existing resources rather than inflating costs.

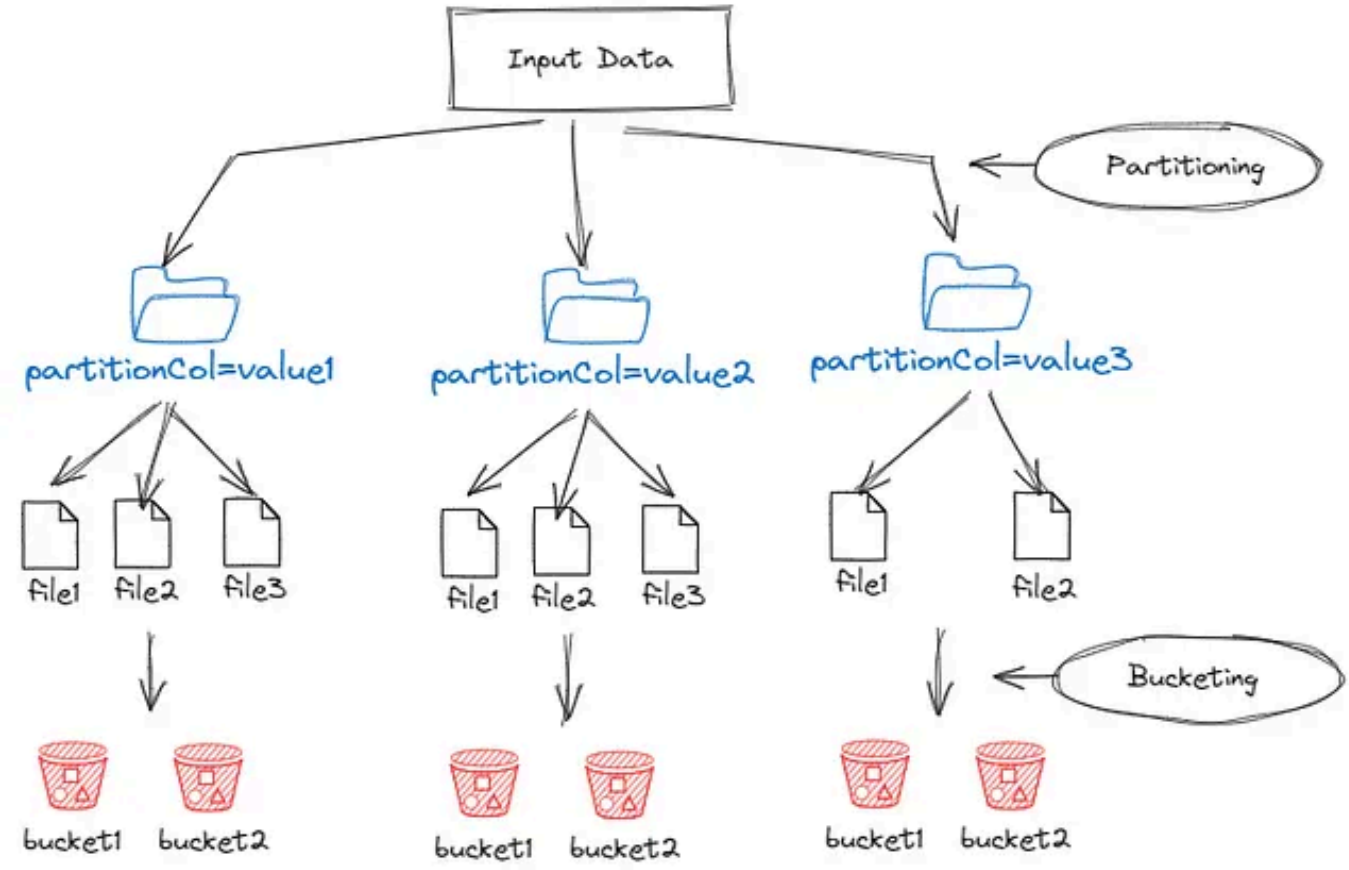
The Lightbulb Moment: It's All About Balanced Buckets

After analyzing our pipeline bottlenecks, we zeroed in on a key issue: the way our partitioned tables were distributed during joins. Our Spark jobs frequently joined massive tables like `Customers` and `Orders`, which were partitioned by date or region. However, the number of partitions (or files) in each table wasn't aligned, leading to inefficient joins and data skew. For example, if `Customers` had 16 partitions but `Orders` had 25, Spark had to shuffle data unevenly across the cluster, causing some executors to process more data than others—a classic recipe for slowdowns.

The solution? Redistribute these tables into a balanced number of buckets, ensuring that the number of partitions in joined tables was either equal or a multiple of each other. This would minimize shuffling, reduce workload imbalance, and make our joins faster and more efficient.

The Solution: A Simple Excel Map and Bucketing Strategy

Here's how we executed the solution:



Step 1: Map Out the Joins in Excel

We started by creating an Excel spreadsheet to catalog all our high-volume tables and their join relationships. The spreadsheet included:

- **Table Names**: The primary tables involved in frequent joins (e.g., `Customers`, `Orders`, `Products`).
- **Partition Keys**: The columns used for partitioning (e.g., `order_date` for `Orders`, `signup_date` for `Customers`).
- **Current Partition Count**: The number of files or partitions per table (e.g., `Customers` had 16 partitions, `Orders` had 25).
- **Join Relationships**: Which tables joined with which (e.g., `Customers` joins `Orders` on `customer_id`).

- **Data Volume:** Approximate size of each table to prioritize optimization for the largest ones.

This map gave us a clear picture of where imbalances existed and which joins were causing the most trouble.

Step 2: Define the Bucketing Strategy

Our goal was to ensure that tables involved in joins had either the same number of partitions or a multiple thereof. For example:

- If `Customers` had 16 partitions, `Orders` should have 16, 32, 48, or another multiple of 16 partitions.
- This alignment ensures that Spark can distribute the data evenly across executors during joins, minimizing shuffling and skew.

To determine the ideal number of buckets, we considered:

- **Cluster Size**: Our cluster had 16 executors, so we aimed for a partition count that was a multiple of 16 to fully utilize all executors.
- **Data Volume**: Larger tables need more partitions to keep each partition manageable (e.g., ~1GB per partition).
- **Join Patterns**: We prioritized tables with frequent joins to ensure compatibility.

After analysis, we settled on 32 buckets as a baseline for most tables, as it was a multiple of our executor count and allowed flexibility for larger tables to scale to 64 or 96 buckets if needed.

Step 3: Repartition the Tables

Using Spark, we repartitioned our tables into the target number of buckets. For example:

- `Customers` was repartitioned from 16 to 32 buckets using `df.repartition(32, "customer_id")` to ensure even distribution based on the join key.
- `Orders` was already at 25 partitions, so we repartitioned it to 32 buckets as well, aligning it with `Customers`.

We saved the repartitioned tables in Parquet format, partitioned by the original key (e.g., `order_date`), with the new bucket count applied. This step was a one-time operation for existing data, and we updated our ingestion pipelines to write new data with the correct number of buckets going forward.

Step 4: Update the Join Logic in Jobs

With the tables now in balanced buckets, we updated our Spark jobs to take advantage of this structure. Since `Customers` and `Orders` both had 32 buckets, Spark could perform the join more efficiently, with minimal shuffling. We also enabled Spark's **Adaptive Query Execution (AQE)** to further optimize the join execution plan dynamically.

The Impact: Faster Jobs, Happier Teams

The results were transformative:

- **50% Faster Job Runtimes:** Jobs that previously took 60 minutes were now completing in 30 minutes, thanks to reduced shuffling and balanced

workloads.

- **On-Time Pipelines**: With faster jobs, our pipelines met their SLAs, ensuring downstream reports were refreshed on schedule.
- **Cost Stability**: We avoided scaling up the cluster, keeping compute costs in check while improving performance.
- **Reduced Resource Contention**: Balanced workloads meant fewer spikes in resource usage, allowing more jobs to run concurrently without bottlenecks.
- **Improved Team Morale**: Analysts got their data on time, and engineers spent less time troubleshooting performance issues.

Why This Worked (and Why You Should Try It)

This approach worked because it tackled the root cause — inefficient joins due to imbalanced partitions — rather than masking the problem with more resources. By aligning the number of buckets across joined tables, we ensured Spark could distribute the workload evenly, minimizing data skew and shuffling. The simplicity of the solution (an Excel map and a one-time repartition) made it easy to implement and maintain, proving that sometimes the simplest solutions yield the biggest wins.

How to replicate this in your environment:

1. **Map Your Joins**: Use a spreadsheet to catalog your high-volume tables, their partition counts, and join relationships.
2. **Define Bucket Rules**: Choose a number of buckets that aligns with your cluster size and ensures joined tables have equal or multiple partition counts.

3. **Repartition Tables:** Use Spark to repartition tables into the target number of buckets, saving them in an efficient format like Parquet.
4. **Optimize Joins:** Update your jobs to leverage the new structure and enable Spark features like AQE for additional performance gains.
5. **Monitor and Iterate:** Track job performance metrics (e.g., runtime, shuffle data size) and adjust bucket counts as data grows.

The Bigger Picture

In data engineering, performance optimization doesn't always require complex solutions or expensive infrastructure upgrades. By focusing on how data is distributed and processed, we achieved significant gains with minimal cost. Redistributing partitioned tables into balanced buckets not only improved our job performance but also set a foundation for scalable growth as our data continues to expand. 🚀

Data Engineering | Data Engineer | Hive | Data Analysis | Spark

Written by Anurag Sharma

83 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!