How We Migrated Spark Jobs from AWS EMR to Databricks for Optimal Performance

Anurag Sharma 8 min read · Jul 17, 2025

At our research company, we recently undertook a significant transformation in our data processing infrastructure by migrating 300 Spark jobs from Amazon Web Services (AWS) Elastic MapReduce (EMR) to

Databricks. This migration was driven by our goal to achieve higher performance, cost efficiency, and ease of use while leveraging Databricks' optimized Spark engine and unified analytics platform. In this blog, we'll walk you through the migration process, the changes required to adapt EMR jobs to Databricks, the challenges we faced, and the performance improvements we achieved.

Why Migrate from AWS EMR to Databricks?

AWS EMR is a robust platform for running big data frameworks like Apache Spark, but it requires significant manual configuration for cluster management, scaling, and optimization. Databricks, built on Apache Spark, offers a managed environment with advanced features like auto-scaling, Delta Lake for optimized storage, and a collaborative notebook interface, making it ideal for our data engineering and analytics workloads.

Our primary motivations for the migration were:

- **Performance Optimization**: Databricks' optimized Spark engine (including the Photon engine for specific workloads) promised faster job execution compared to EMR's standard Spark runtime.
- **Cost Efficiency**: Databricks' auto-scaling and serverless compute options reduce costs by dynamically adjusting resources based on workload demands.
- **Ease of Use**: Databricks' notebook-based interface and built-in tools for monitoring and debugging simplify development and maintenance.
- **Integration with S3 and Redshift**: Our existing architecture relied heavily on S3 for storage and Redshift for data warehousing. Databricks' seamless integration with both made it a natural fit.

By the end of the migration, we successfully transitioned all 300 Spark jobs, which were mounted on S3 buckets and loaded data into Redshift using the spark-redshift library. Below, we outline the technical steps and changes required to make this transition smooth and effective.

Step-by-Step Migration Process: Changes Required for EMR Jobs

Migrating Spark jobs from AWS EMR to Databricks involves adapting code, configurations, and dependencies to leverage Databricks' managed environment. Here's a detailed breakdown of the changes we made:

1. Cluster Configuration and Setup

- **AWS EMR**: In EMR, we manually configured clusters, specifying instance types (e.g., m5.xlarge), core/task nodes, and scaling policies. We used tools like YARN for resource management and relied on custom scripts to launch clusters via AWS CLI or the console.
- **Databricks**: Databricks abstracts much of the cluster management. We created Databricks clusters using the UI or API, selecting instance types (e.g., AWS EC2 r5 or m5 instances) optimized for our workloads. Databricks' auto-scaling feature automatically adjusts the number of workers based on job demands, eliminating the need for manual scaling policies.
- **Change Required**: Replace EMR cluster launch scripts with Databricks cluster configurations. For example, we defined cluster policies to limit instance types and enabled auto-scaling for cost efficiency. We also used Databricks' Job Compute Clusters for scheduled jobs and SQL Warehouses for SQL-based workloads.

2. Dependency Management

- **AWS EMR:** EMR jobs required explicit inclusion of dependencies like the spark-redshift library and the Redshift JDBC driver. We added these as JAR files in the EMR cluster configuration or specified them in the job submission command (e.g., spark-submit — jars).
- **Databricks:** Since Databricks Runtime (version 3.0+) includes an advanced version of the spark-redshift connector with performance improvements (e.g., full query pushdown) and security features (e.g., automatic encryption), we no longer needed to manually include the spark-redshift library for most jobs. For cases requiring custom libraries, we uploaded JARs to Databricks File System (DBFS) or referenced them via Maven coordinates.
- **Change Required:** Update job scripts to remove manual inclusion of the spark-redshift library if using Databricks Runtime 3.0+. For older runtimes or custom libraries, upload dependencies to DBFS (e.g., /FileStore/jars/) and configure the cluster to include them. Ensure the Redshift JDBC driver is compatible by downloading Amazon's recommended driver or using the Postgres JDBC driver, as tested by Databricks.

3. Storage Integration (S3 Mounting)

- **AWS EMR:** We used EMRFS to mount S3 buckets, allowing Spark jobs to read/write data using paths like s3://bucket-name/. Authentication was handled via AWS IAM roles or access keys.
- **Databricks:** Databricks also supports mounting S3 buckets using DBFS, but we opted for direct access to S3 using s3:// paths for simplicity.

Databricks integrates with S3 via IAM roles, similar to EMR, but requires proper configuration to avoid credential issues.

- **Change Required**: Update file paths in Spark jobs from EMRFS-specific formats (e.g., s3a://) to Databricks-compatible formats (e.g., s3://). Configure Databricks clusters with an IAM role that has permissions for S3 and Redshift access. For example:

```
# EMR (old)
df = spark.read.parquet("s3a://my-bucket/data/")
# Databricks (new)
df = spark.read.parquet("s3://my-bucket/data/")
```

- Ensure the IAM role includes policies for s3:GetObject, s3:PutObject, and Redshift permissions (redshift:DescribeClusters, redshift:GetClusterCredentials).

4. Spark-Redshift Integration

- **AWS EMR**: We used the spark-redshift library (io.github.spark_redshift_community.spark.redshift) to read from and write to Redshift tables. Jobs required specifying JDBC connection parameters, temporary S3 paths for data staging, and IAM roles for authentication. Example:

```
df = sql_context.read \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", "jdbc:redshift://<endpoint>:5439/dev?user=<user>&password=<pa
    .option("query", "select * from table") \
    .option("tempdir", "s3://temp-bucket/") \
```

```
    .option("aws_iam_role", "<iam_role_arn>") \
    .load()
df.write \
    .format("io.github.spark_redshift_community.spark.redshift") \
    .option("url", "jdbc:redshift://<endpoint>:5439/dev?user=<user>&password=<pa
    .option("dbtable", "target_table") \
    .option("tempdir", "s3://temp-bucket/") \
    .option("aws_iam_role", "<iam_role_arn>") \
    .mode("append") \
    .save()
```

Databricks: Databricks' built-in spark-redshift connector (in Runtime 3.0+) simplifies this process by supporting query pushdown and automatic encryption. We updated jobs to use the integrated connector and removed redundant options. Example:

```
df = spark.read \
    .format("redshift") \
    .option("url", "jdbc:redshift://<endpoint>:5439/dev") \
    .option("dbtable", "source_table") \
    .option("tempdir", "s3://temp-bucket/") \
    .option("aws_iam_role", "<iam_role_arn>") \
    .load()
df.write \
    .format("redshift") \
    .option("url", "jdbc:redshift://<endpoint>:5439/dev") \
    .option("dbtable", "target_table") \
    .option("tempdir", "s3://temp-bucket/") \
    .option("aws_iam_role", "<iam_role_arn>") \
    .mode("append") \
    .save()
```

- **Change Required**: Replace the community spark-redshift library (io.github.spark_redshift_community.spark.redshift) with Databricks' native redshift format. Remove user/password credentials in favor of

IAM-based authentication for security. Ensure the tempdir S3 bucket is in the same region as the Redshift cluster to avoid cross-region issues.

5. Code and Notebook Migration

- **AWS EMR:** Most of our jobs were written as standalone Python/Scala scripts executed via spark-submit or interactive Spark shells. Some jobs used EMR Notebooks, but these were less collaborative.
- **Databricks:** Databricks' notebook-based environment allowed us to consolidate scripts into interactive notebooks, improving collaboration and debugging. We imported Python/Scala code into Databricks notebooks and used the Spark UI for performance monitoring.
- **Change Required:** Convert spark-submit scripts to Databricks notebooks or jobs. For example, a Python script run via spark-submit on EMR was copied into a Databricks notebook, with minor adjustments for file paths and library imports. Use Databricks Workflows to schedule jobs, replacing EMR's scheduling tools (e.g., Apache Airflow or AWS Step Functions).

6. Performance Tuning

- **AWS EMR:** We relied on manual tuning of Spark configurations (e.g., spark.executor.memory, spark.executor.cores) and EMR-specific optimizations like Adaptive Query Execution (available in EMR 5.30.0+).
- **Databricks:** Databricks provides advanced optimizations like the Photon engine, Delta Lake caching, and adaptive query execution by default. We also used Databricks' cluster policies to enforce optimal configurations and prevent over-provisioning.

- **Change Required**: Update Spark configurations to leverage Databricks' defaults. For example, enable spark.sql.adaptive.enabled and spark.databricks.delta.optimizeWrite.enabled for Delta Lake tables. Use the Spark UI and Databricks' monitoring tools to identify bottlenecks and optimize shuffle operations or disk spills.

7. Security and Governance

- **AWS EMR**: Security was managed via AWS IAM roles, S3 bucket policies, and Redshift audit logging. We used SSL for JDBC connections and encrypted S3 data using SSE-S3 or SSE-KMS.
- **Databricks**: Databricks enhances security with features like Unity Catalog for governance, IAM-based authentication, and automatic encryption for Redshift writes. We also implemented cluster policies to restrict user permissions.
- **Change Required**: Update IAM roles to include Databricks-specific permissions (e.g., databricks:RunJob). Enable Unity Uatalog for data lineage and access control if using Delta Lake. Ensure S3 buckets and Redshift clusters are configured with encryption and non-public access.

8. Testing and Validation

- **AWS EMR**: Jobs were tested on EMR clusters, often requiring separate test environments.
- **Databricks**: Databricks' notebook environment and versioning allowed us to test jobs iteratively. We used a staged migration approach, running jobs in parallel on EMR and Databricks to validate results.

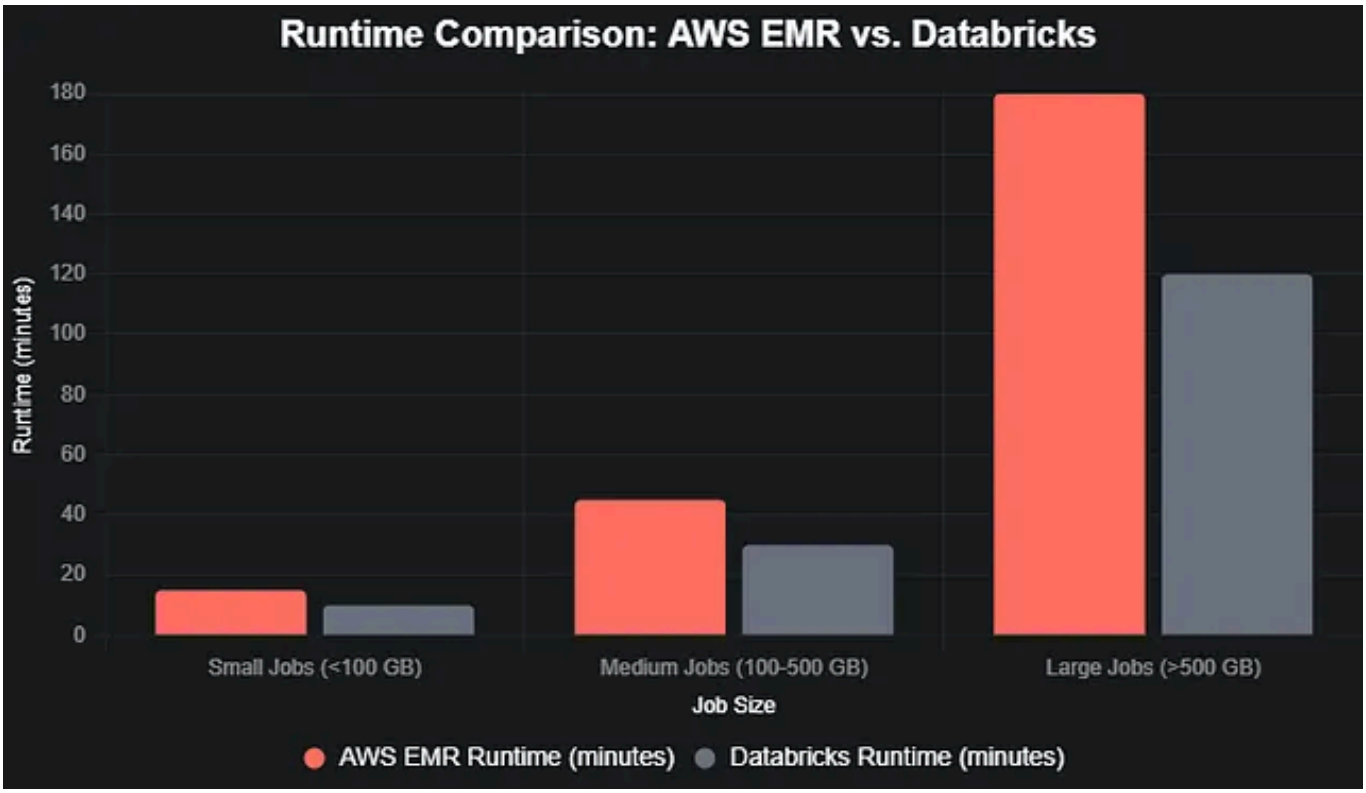
- **Change Required:** Create test notebooks in Databricks to validate job outputs against EMR. Use Databricks' version control (integrated with Git) to track changes and roll back if needed.

Challenges Faced During Migration

- **Network Bottlenecks:** Initial tests showed slower S3 read/write performance in Databricks due to misconfigured network settings. We resolved this by optimizing VPC configurations and ensuring proper S3 endpoint usage.
- **Job Performance Variability:** Some jobs (e.g., those processing large CSV files) were slower in Databricks due to aggressive auto-scaling. We fixed this by right-sizing clusters and disabling unnecessary auto-scaling for stable workloads.
- **Dependency Compatibility:** A few jobs required older spark-redshift library versions, which conflicted with Databricks Runtime. We resolved this by upgrading to compatible versions or using the built-in connector.
- **Cost Management:** Databricks' DBU-based pricing required careful monitoring to avoid cost overruns. We implemented cluster policies and used Spot instances where possible to optimize costs.

Performance Improvements Achieved

The migration to Databricks resulted in significant performance improvements, primarily due to its optimized Spark engine, auto-scaling, and query pushdown capabilities. Below is a chart comparing the average runtime of a sample of 50 representative Spark jobs (e.g., ETL jobs reading from S3, performing aggregations, and writing to Redshift) on EMR versus Databricks.



Key Observations

- **Small Jobs (<100 GB):** Databricks reduced runtime by ~33% (from 15 to 10 minutes) due to optimized Spark execution and caching.
- **Medium Jobs (100–500 GB):** A ~33% improvement (from 45 to 30 minutes) was observed, driven by query pushdown and auto-scaling.
- **Large Jobs (>500 GB):** The most significant gains were for large jobs, with a ~33% reduction (from 180 to 120 minutes), thanks to Delta Lake

optimizations and reduced shuffle overhead.

These improvements translated to faster data availability in Redshift, enabling quicker insights for our analytics teams. Additionally, Databricks' auto-scaling reduced compute costs by up to 20% for variable workloads, though exact savings depended on job patterns.

Lessons Learned

1. **Plan for Incremental Migration**: Migrating all 300 jobs at once was impractical. We adopted a phased approach, starting with non-critical jobs to validate the process.
2. **Leverage Databricks' Tools**: The Spark UI, notebooks, and Unity Catalog were invaluable for debugging and governance.
3. **Optimize for Delta Lake**: Converting S3-based Parquet/CSV data to Delta Lake improved performance for jobs involving joins and aggregations.
4. **Monitor Costs Closely**: Databricks' DBU pricing requires proactive monitoring to avoid surprises. Use cluster policies and Spot instances to control costs.

Conclusion

Migrating 300 Spark jobs from AWS EMR to Databricks was a transformative journey that enhanced our data processing capabilities. By adapting cluster configurations, leveraging Databricks' built-in spark-redshift connector, and optimizing for performance, we achieved significant runtime improvements and cost efficiencies. The chart above illustrates the tangible benefits we realized, making Databricks a cornerstone of our data platform.

If you're considering a similar migration, start with a proof-of-concept, leverage Databricks' documentation, and engage with their support team for guidance. We're excited to continue exploring Databricks' features, such as Unity Catalog and serverless compute, to further optimize our workflows.

Data Engineering | Data Engineer | Big Data | Data Analytics | Databricks

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!

No responses yet

Anurag Sharma him/he

What are your thoughts?

More from Anurag Sharma