Optimizing Spark Jobs: Solving Upstream Data Quality Issues with a Custom DQ Framework

Anurag Sharma 5 min read · Jan 5, 2026

In the fast-paced world of data engineering, few things are as frustrating as watching your meticulously crafted Spark jobs crumble due to faulty upstream data. At our organization, we were dealing with this nightmare daily. With over 400 tables ingested from various sources, inconsistencies like missing values, incorrect entries, duplicates, and outliers were rampant. These issues did not just cause job failures; they led to data corruption in our reporting pipelines, forcing us into a cycle of manual fixes. In this blog, I will walk you through how we turned this chaos into a streamlined process by building a robust Data Quality (DQ) framework.

The Problem: A Daily Battle with Bad Data

As the consumption and reporting team, we own the silver and gold layers, the final frontiers for accurate, reliable data used in business reporting and analytics. Upstream teams deliver raw data to our landing zones in S3, but all too often, that data arrives riddled with problems:

- **Missing Values**: Critical fields left blank, breaking downstream transformations.
- **Incorrect Data**: Mismatched formats, invalid entries, or outright errors.
- **Duplicates**: Redundant records inflate datasets and skew aggregates.
- **Outliers**: Extreme values that could indicate deeper issues, but often just cause failures.

When these hit our Spark-based curation pipelines, the results were predictable: jobs failed, data got corrupted, and we would scramble to:

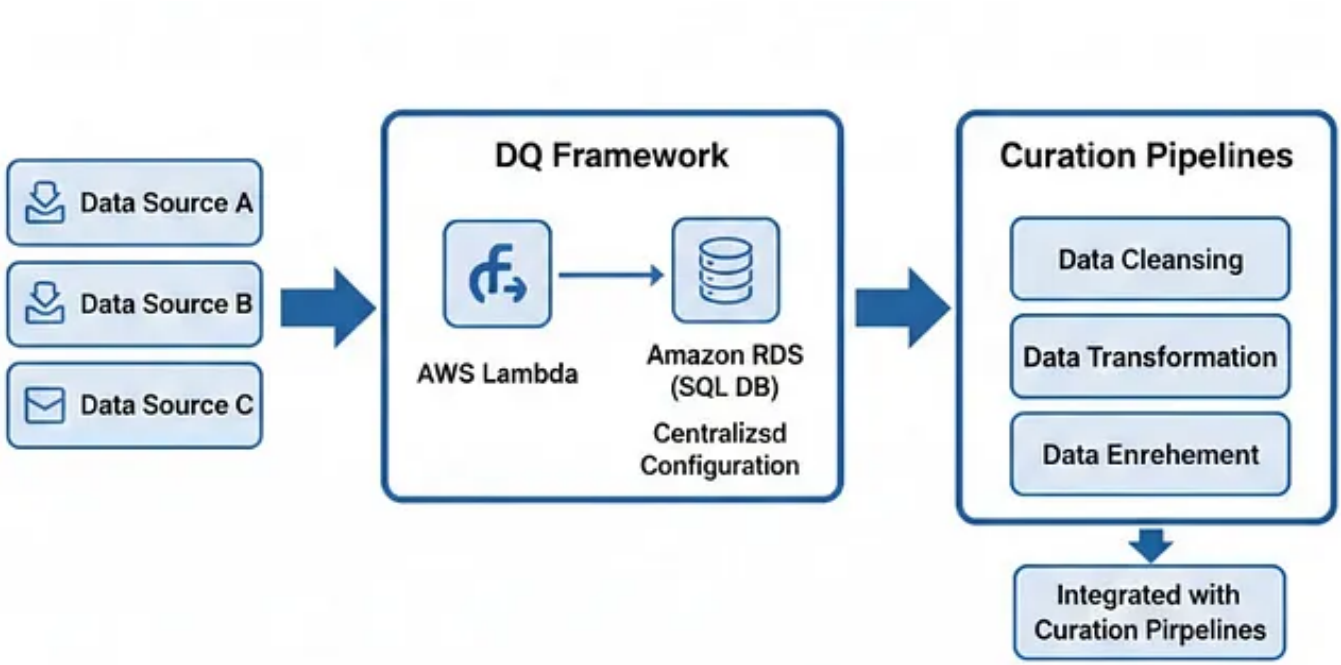
1. Re-run the job using older, stable data.
2. Delete the latest corrupt entries from our tables.
3. Fire off emails to upstream teams, begging them to fix their side.

This was not just inefficient; it was soul-crushing for developers. Manual interventions in the silver and gold layers are risky; one wrong move could cascade into flawed reports, eroding trust in our data ecosystem. We needed a proactive shield against bad data, not a reactive Band-Aid.

Our Solution: A Pre-Consumption DQ Framework

We decided to flip the script by implementing a comprehensive Data Quality framework right at the ingestion point. The goal? Validate data before it ever touches our curation pipelines. Since we are downstream consumers, we could not control the source quality, but we could enforce checks on what we accept.

The framework acts as a gatekeeper: It samples incoming data, runs targeted queries, and only promotes "clean" data to the raw layer for further processing. If issues are detected, the data stays quarantined, and alerts go out automatically — saving developers from daily firefighting.



Building the Framework: Step by Step

Here is how we architected and implemented this system using AWS services for scalability and cost-efficiency.

1. Centralized Configuration with AWS RDS (SQL DB)

We started by creating a dedicated table in an AWS RDS instance (using PostgreSQL for simplicity). This table serves as the brain of the operation, storing:

- **S3 File Locations**: Paths to the landing zones where upstream data arrives (e.g., s3://bucket/landing/table_name/).
- **DQ Queries**: Custom SQL queries tailored to each table's schema and business rules.

Examples include:

- Checking for nulls: SELECT COUNT(*) FROM table WHERE critical_column IS NULL;
- Detecting duplicates: SELECT COUNT(*) FROM (SELECT id, COUNT(*) FROM table GROUP BY id HAVING COUNT(*) > 1) AS dupes;
- Validating ranges: SELECT COUNT(*) FROM table WHERE value < 0 OR value > 100;
- Outlier detection: Basic statistical checks using percentiles or z-scores.

This setup allows us to easily add, update, or version queries without redeploying code, perfect for evolving data requirements.

2. Serverless Processing with AWS Lambda

The core engine is an AWS Lambda function, triggered on data arrival (via S3 event notifications for batch checks).

Here is the workflow:

- **Random Sampling for Efficiency**: Instead of scanning the entire dataset (which could be massive and costly), Lambda randomly selects 2–3 files from the S3 landing location. This subset represents a probabilistic sample, giving us a high-confidence check without the overhead of full scans.
- **Query Execution**: Using libraries like PyAthena, we load the sample into a temporary Athena table and run the predefined DQ queries.
- **Validation and Promotion**: If all queries pass (e.g., no violations above thresholds), Lambda uses aws s3 sync or aws s3 cp to move the files from the landing bucket to the raw bucket. If any query fails, the data stays in landing, an alert is sent (via SNS to upstream teams and our Slack channel), and the process halts.

This sampling approach gives us an "edge" in performance: We catch most issues early without exhaustive processing, keeping costs low and speed high.

3. Integrating with Curation Pipelines

Our existing Spark jobs handle curation from raw to silver/gold layers, running either incrementally (for append-only tables) or as full loads (for overwrites).

To accommodate the DQ step:

- We shifted the scheduling of these jobs by 1 hour ahead. This buffer ensures the DQ checks complete first, reducing the chance of encountering bad data mid-process.
- Post-DQ, if data is promoted to raw, the curation jobs trigger automatically via Airflow, maintaining our end-to-end pipeline.

The Impact: From Chaos to Control

Implementing this DQ framework transformed our operations:

- **Reduced Failures**: Spark job failures dropped by over 70%, as bad data is caught upstream.
- **Time Savings**: Developers no longer spend hours on manual fixes — automation handles detection and notification.
- **Improved Data Trust**: With proactive checks, our silver and gold layers are more reliable, leading to fewer downstream issues in reporting.
- **Cost Efficiency**: Sampling minimizes compute usage; we only process full datasets when they're validated.
- **Scalability**: Handles our 400+ tables effortlessly, with easy extensibility for new sources.

Conclusion: Empowering Data Teams with Quality Gates

In data engineering, quality is not a luxury; it's a necessity. By building this DQ framework, we not only solved our Spark job woes but also fostered better collaboration with upstream teams.

If you're facing similar upstream data headaches, consider starting small:

> Identify your pain-point tables -> Define key checks -> Automate with serverless tools

Spark | Data Quality | AWS | AWS Lambda | Automation

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