International Journal of Artificial Intelligence, Data Science, and Machine Learning
Grace Horizon Publication | Volume 6, Issue 1, 174 -185, 2025
ISSN: 3050-9262 | https://doi.org/10.63282/3050-9262.IJAIDSML-V6I1P119

Original Article

From SQL to Spark: My Journey into Big Data and Scalable Systems How I Debug Complex Issues in Large Codebases

Bhavitha Guntupalli
ETL/Data Warehouse Developer at Blue Cross Blue Shield of Illinois, USA.

Received On: 17/12/2024 ***Revised On: 27/12/2024*** ***Accepted On: 19/01/2025*** ***Published On: 05/02/2025***

Abstract - *From a well-lit workshop to the vast, dynamic environment of modern big data systems is like traversing SQL to Apache Spark. Beginning in the familiar realm of relational databases, regulated information and careful searches dictated my road forward. But as batch projects needing hoursor even days started to demand growing data volumes, I realized that traditional SQL systems were insufficient. At that time I asked Spark for help. The transformation was not erratic. I started to rethink data pipelines, fault tolerance, and distributed computing. Debugging challenging problems in large-scale Spark systems posed hitherto unmet challenges including tracking lineage using Directed Acyclic Graphs (DAGs), memory spill management, and cluster performance optimization. I spent days reviewing Spark UI logs to find how little coding mistakes may seriously affect performance. These encounters sharpened my intuition and made it obvious that scalable systems demand scalable thinking: build with possible failure in mind, break out jobs for parallel execution, and often check what you produce. The most important lesson was changing perspectivefrom seeing data operations as individual SQL queries to seeing they were a part of a dynamic, strong architecture. Today, my Spark path has enhanced my technological knowledge as well as my engineering abilities in terms of handling uncertainty, scalability, and simplicity-based change. This links to the development of our cognitive processes in a period when data grows at hitherto unheard-of speed, transcending simple tool replacement.*

Keywords - *Big Data, SQL, Apache Spark, Scalable Systems, Debugging, Distributed Computing, Data Engineering, Codebase Complexity, Performance Optimization.*

1. Introduction

SQL has always been the universal tongue in data engineering. Analytics has been developed for many years using traditional SQL-based technologies, which have enabled ETL pipelines and running business intelligence dashboards. Usually based on relational databases like MySQL or Postgres or data warehouses like Oracle and Teradata, these solutions gave dependable performance and simplicity for structured data and moderate workloads. SQL lets data engineers and analysts create manageable, unambiguous declarative logic for analysis, purification, and conversion of data. As businesses evolved and their online presence grew, the shortcomings of these conventional technologies became ever more obvious. The increasing data volume, diversity, and speed have profoundly changed the topography. Instead of gigabytes, companies were managing terabytes and petabytes of semi-structured or unstructured data coming from various sourcesIoT devices, application logs, social media feeds, and others. While batch processing with overnight jobs was inadequate, stakeholders demanded real-time information, instantaneous alarms, and dashboards displaying current events rather than historical summaries. Growing needs revealed the scalability limits of monolithic SQL-based systems, which led to a significant change towards distributed computing and contemporary data platforms like Apache Spark.

This move presents new difficulties not only related to tool exchange but also a question of perspective. Among the most challenging jobs accessible is navigating and debugging big, scattered codebases. Unlike traditional SQL scripts in which faults are usually localized and repeatable individually, debugging in big data systems includes negotiating execution plans spread across clusters, concealed dependencies, data skew, memory management concerns, and non-deterministic failures. Particularly for programmers transitioning from SQL to Scala, PySpark, or streaming models using Spark Structured Streaming or Apache Kafka, the learning curve could be steep. The initial experience of a practitioner traversing the familiar SQL domain to the complex and strong domain of Spark-based scalable systems is attempted to be described in this work. This will examine the motivations for applying big data technologies, underline the technical and conceptual difficulties faced, and provide feasible solutions for main data pipeline flaws. By means of practical ideas and real-world examples, this article seeks to elucidate the strategy, thereby giving engineers a pragmatic viewpoint on a comparable road.

This content serves as both a guide and a source of inspiration, whether your thoughts are about the relocation or whether you are actively working on Spark chores and cluster logs. Different concepts abound in scalable systems, and knowing their navigation calls for not only a philosophical but also a technical transformation.

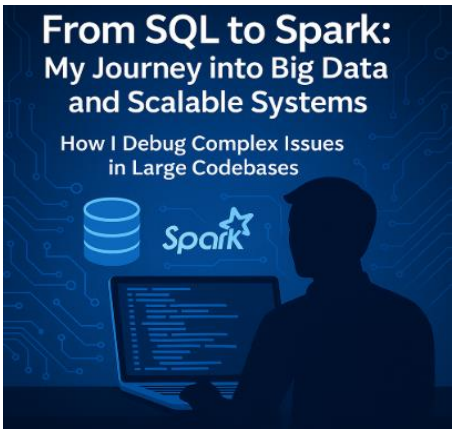

Figure 1: Debugging Complex Issues in Scalable Big Data Systems: From SQL to Spark

2. The SQL Era: Simplicity, Strengths, and Limitations

For many data professionals, the first step into analytics starts with SQL. It's simple, clear, and strong. SQL's declarative structure helps analysts and engineers focus on the intended data retrieval or transformation instead of the computing effort the system demands. Together with decades of growth, this abstraction helped SQL to become the main language for handling and searching structured data.

2.1. SQL-Based Architectures: The Classic Model

Conventional SQL-based systems were developed using Oracle, SQL Server, Teradata, ormore recentlycloud-native replacements like Amazon Redshift and Google Big Query around a central data warehouse. First placed into staging tables, raw data underwent a sequence of changesjoins, filters, and aggregationsdone via planned batch processes. Usually working overnight to ensure reports and dashboards were ready for the next business day, these were done using lightweight schedulers like Airflow or DBT or cron tasks. These tools were primarily batch-oriented. Regular daily pipelines using ETL solutions such as Informatica or Talend could compile data from transactional systems, convert the data through intricate SQL scripts, and then place the final results into analytics-ready tables. This approach performed effectively for expected workloads based on either stagnant or progressively changing data.

2.2. Strengths of the SQL Approach

Going back to the days of SQL, this technology was the best known to offer certain advantages that made it the go-to data infrastructure for the beginning of the era:

- **Simplicity and Readability:** SQL is not hard to understand and to read. It has a syntax that is very close to natural language; thus, it is understandable also by non-engineers.
- **Mature Ecosystem:** Tools and practices around SQL have matured for decades. Almost every BI tool (Tableau, Power BI, Looker, etc.) has no problem integrating with SQL backends.
- **Robust Tooling and Support:** In addition to schema design, indexing and query optimization, SQL engines offer various features and a rich experience that has been tested and improved through years of use.
- **Transaction Support and ACID Guarantees: S**QL-based databases are unrivaled in their ability to ensure data integrity, which is a key characteristic of financial and operational systems.

These qualities helped businesses to base their decisions on data with minimal engineering effort. Teams could get by with a few well-written scripts and scheduled jobs to deliver insights and maintain data hygiene.

2.3. Limitations in the Age of Big Data

However, as organizations have accumulated huge amounts of data and have required deeper insights at a faster pace, the weaknesses of SQL-based systems have become more pronounced.

- **Vertical Scaling Constraints**: More traditional databases are scaled vertically, which implies that they can only gain the capacity to process more by substituting the existing machines with ones of higher power (and more expensive). This concept has financial and practical restrictions.
- **Inefficiency with Large Datasets:** Complex joins, particularly if the tables are wide or huge, frequently lead to long runtimes. Sorting, grouping or aggregating a large amount of data not only drains the system resources but also can cause the pipeline to run very slowly.
- **Long and Fragile Batch Jobs:** Many ETL pipelines depend on sequential dependencies for their operations. If one cron job disappears, the whole chain will get broken, and the report will not be delivered on time, which will upset the stakeholders.
- **Lack of Real-Time Capabilities:** SQL systems did not have streaming or real-time processing as their core functionalities during their creation. If anyone wanted to have fresh and on-demand insights, then they were forced to use awkward workarounds or pay a lot for the change of the architecture.
- **Operational Blind Spots:** SQL scripts are easily readable but they do not have built-in observability. To find out where the slowness or the lack of data in some parts comes from, one needs to dig deeply into logs or monitor dashboards.

2.4. Case Example: Marketing Attribution Pipeline

One especially memorable incident from my past pertains to a marketing attribution project for an internet shop. Every transaction had to be assigned to the appropriate mix of user touchpointsemail marketing, social media contacts, referral programs, etc. Unprocessed data found homes in transactional databases, site logs, and email marketing systems. We merged this data using a multi-stage SQL pipeline. Cron duties watched over the pipeline located on a Postgres data warehouse. After managing several thousand daily transactions, the work took twenty minutes initially. As the business expanded and daily transaction totals surpassed five million rows, the job began to last several hours. Join operations using huge session and clickstream tables produced disk leakage and lock contention even with indexes and query optimization. Moreover, the restricted observability made the identification of bottlenecks and such presumption-based difficulties difficult. The performance dashboards utilized by sales and marketing leadership suffered as the work routinely fell short of the morning reporting Service Level Agreement. The rigidity of the system discouraged iteration. Every changeincluding changing attribution logicforced the rebuilding of multiple closely related SQL searches. Analyzing those modifications required repeating pipeline segments, copying production data, and depending on no downstream table failures during the process. Our best efforts notwithstanding, the architecture had run its course.

2.5. Lessons from the SQL Era

Like many others, this study showed that although SQL-based systems are ideal for fast prototyping and limited scalability, they struggle when required to process huge volumes or provide real-time analysis. Their preferred monolithic techniques, centralized execution, and strong coupling turn unfavorable as size and complexity rise. The knowledge let me appreciate distributed execution engines, modular design, and improved debugging tools. These revelations finally inspired me towards Spark and big data ecosystems, where scalability and observability define the basic architecture. One would naturally find the SQL age to be straightforward. It certainly helped me to understand data modeling, transformation thinking, and the need for clarity. However, it also underscored the shortcomings of trying to expand existing systems outside their natural limitationsa knowledge that motivated research into networked, scalable data systems.

3. Enter Apache Spark: Embracing Distributed Processing

About the scalability constraints of traditional SQL systems, I looked for a platform able to handle the increasing complexity of our projects and data volume. By then Apache Spark was a distributed processing engine painstakingly developed for best speed, scalability, and fault tolerance. Spark first seemed difficult because of its own paradigms and programming interfaces; yet, it provided a capacity not accessible with SQL alone: consistent and quick processing of enormous volumes of data over a cluster of machines.

3.1. Introduction to Spark: Core Concepts

Once I understood Spark's three main ideas, they totally revolutionized my mindset about data processing:

- **Resilient Distributed Datasets (RDDs):** RDDs in Spark are the core; they are collections of objects that are distributed and immutable. They allowed active control over the conversions and the operations between nodes; however, the cost was increased syntactic complexity and the need for human optimization.
- **DataFrames and Datasets:** These APIs of high order provide a more organized and declarative way for the data interaction from DataFrames and Datasets, e.g., SQL or pandas. They permitted improvements by means of the Catalyst query optimizer for the majority of analytical operations, therefore generating more output performance than bare RDDs.
- **Directed Acyclic Graphs (DAGs):** Spark creates a DAG of execution based on the logical framework of the transformations instead of actually doing each one immediately. Spark can make the best use of the going strategy by exploiting this lazy evaluation approach, thus lessening the shuffles and focusing the processes for the best performance.
- **Lazy Evaluation:** In contrast to SQL or the usual imperative programming, Spark does not process anything until an action is performed, for instance, collect(), write(), or count(). At first, this method was confusing; however, it turns out to be the best way Spark saves energy and increases effectiveness.

In sum, these components affected the very concept of designing a data pipeline. From SQL to Spark, the change was definitely not simply a change of the language, but it was also a change in my conceptual understanding of data flow, memory handling, and execution strategies.

3.2. Adoption Curve: Resistance, Learning, and Breakthroughs

Right away, my response to Spark was conflicted. Its prospects excited me on one side; the difficulty of remote debugging and the unknown APIs terrified me on the other. Unlike SQL, which typically provides a detailed error message for failed searches, Spark activities fail in mysterious ways, including ambiguous stack traces buried in logs, serialization problems, or executor memory failures. The Spark UI appeared as alien an interface with its phases, chores, and DAG representations. Disturbances also caused a loss of productivity. Originally only just a few lines of SQL, simple tasks today need many lines of PySpark. The repl-driven development felt labor-intensive. The feedback loop was slow; evaluating a modification typically meant executing a complete

Directed Acyclic Graph (DAG) and waiting several minutes for results. The pieces started to line up gradually. I began efficiently cutting recomputation with persist() and cache(). I became able to maximize partition sizes, spot and prevent expensive shuffles, and reduce high-scale dependencies. Originally a mystery, the Spark UI developed into a diagnostic tool. Explain(), adaptive query execution (AQE), and broadcast joins provide a more methodical approach to finding performance problems.

3.3. Migration Process: Translating SQL to Spark Pipelines

Reengineering historical SQL-based ETL systems in Spark is one of the toughest challenges. The goal transcended just "converting" the thinking to reconceptualize it using dispersed concepts. Reengineering of a multi-join SQL script enabled with PySpark DataFrames integrating transaction, clickstream, and campaign data. Originally SQL included flattening, aggregate logic demanding sequential execution, and window functions including nested subqueries. Spark came forth in modular stages:

- **Ingestion:** Raw data originated from Parquet-style distributed systems like S3 or HDFS.
- **Transformations:** Every change consisted of a set of choose(), join(), withColumn(), and groupby() instructions. These arrange themselves logically with temporary storage.
- **Aggregation and Enrichment:** Business logic based on attribution principles drove sequential transformations, delaying materialization till needed.
- **Output:** The finished datasets for the next investigation were either returned to Delta Lake or data lakes.

The modular construction improved the testability and scalability of the pipeline. Moreover, Spark's horizontal scalability allowed the same reasoning to manage hundreds of millions of records with little changean attempt inconceivable in our previous SQL systems.

3.4. Key Learnings: Thinking Distributed

The biggest change by far in working with Spark was definitely the way of thinking in parallel. Normally, SQL processing is designed for rows; even if a number of engines work in parallel, the main conceptual framework still works sequentially.

- **Embrace Data Partitioning:** Knowing how Spark shared data among executors became really crucial. Adopt data splitting. Longer runtimes and data skew resulting from less effective partitioning resulted. Techniques of tailored partitioning improved join and aggregation performance.
- **Design for Fault Tolerance:** In fault tolerance architecture, Spark uses retries and automatically controls node failures. This changed my viewpoint of pipeline resilience, checkpointing, and job idempotency.
- **Tune for Performance at Scale:** Mastery of the settings of spark.sql.shuffle.partitions, executor memory, and broadcast thresholds determines almost everything about scalable performance. Just as important now as developing logical reasons is optimizing cluster resources.
- **Decouple Logic into Stages:** Instead of making one large change, I began slicing segment logic into small, verified pieces. This makes the process of realistic fault isolation and component repurposing.

4. Debugging in Large-Scale Systems: Key Challenges

Although Apache Spark presented several difficult and sometimes disruptive debugging problems, the move from traditional SQL systems helped improve processing. Errors in extensively distributed systems are not straightforward. They might show just under particular data volumes, show sporadically, or change deep beneath the execution stack over a worker node network. Over my career, I have found and developed the ability to overcome several obstacles unique to large companies.

4.1. Common Issues in Distributed Data Pipelines

4.1.1. Memory Leaks and OOM Errors

A lot of people have trouble remembering things, which could make it hard to use Spark. Because each executor only has a certain amount of memory, transformations that don't work, like huge aggregations or not enough caching, can quickly run out of memory (OOM). Spark will try to fix these issues by crashing executors, but it can only do this a few times before the job stops working. The main group by key() operation on a field with a lot of data kept making the task fail. Spark wanted to mix up the keys and store them all in memory, but this caused memory leaks and other issues. Using reduceByKey() to combine some of the data before the shuffle saved RAM.

4.1.2. Skewed Data

When one or more partitions have a lot of data, the results are skewed and some procedures take a lot longer than others. This leads to long phases and an uneven allocation of resources. Many "hot" product IDs made it hard to connect the transaction logs and product catalog because they happened millions of times more often than other IDs. Once you saw the skew using the Spark UI and did salting, which means adding random suffixes to skewed keys, the burden was evenly split between the partitions.

4.1.3. Inconsistent Outputs

Distributed systems exhibit nondeterminism depending on other systems, concurrent writing, or asynchronous writing. This ambiguity can lead to difficultly replicable mistakes.

Using operations such as collect() or take() in Spark's production logic could generate inconsistent results unless suitably protected with ordering or deterministic transformations.

4.1.4. Long-Running Stages and DAG Failures

From big changes requiring join, group by, or coalesce, shuffle operations usually follow. Unoptimized variations of these can generate long-standing phases, either failing or stopping the entire process and significant intermediate files. Late in the job, life failures in Directed Acyclic Graphs (DAGs) caused by running out of stage retries particularly irritate me.

4.2. Complexity of the Broader Ecosystem

Indeed, it is correct that Spark is very capable but it is rarely the only component in a data pipeline network in today's world. The data pipelines that are mentioned consist of a number of interconnected systems that have their own interfaces, logs, and failure characteristics:

- Kafka for ingesting real-time event streams
- HDFS/S3 for persistent data storage
- Hive/Delta Lake for metadata and table schema management
- Airflow for DAG orchestration and job dependencies
- Monitoring tools like Datadog, Prometheus, or custom dashboards

The integrations create a larger network of observability, but they also increase the number of places that can cause failures. A small issue that happens in the Kafka topic ingestion can cause a delay in the arrival of data, and because of this, invalid joins may be created, or the report may be incomplete without your realizing it. There is also a possibility that Airflow does not understand the reason for the failure of a Spark job and therefore it continuously retries this until the problem becomes even worse. Monitoring can show high CPU usage, but the mystery of why a particular task is not moving can still be there. Hunting for the problems in this complex setup is not merely a technical job. It is also like being a detective.

4.3. Pain Points in the Debugging Process

- **Logging Granularity:** Although spark logs have a lot of scattered information that might appear unorganized, they are quite exhaustive. At the executor level, errors are then aggregated; therefore, it is not easy to identify the exact part of the transformation that is wrong. Extracting the required information from a gigabyte of files might be a bit challenging.
- **Multilayered Stack Traces:** Debugging information can be very misleading, as failures can be from all sorts of places, such as user-defined library functions, external libraries, or even the cluster manager. The most difficult part of failure is deciding exactly where the problem has occurred by reading the system traces and looking at the system metrics.
- **Reproducibility:** Distributed errors often disappear in small or localized scenarios, making repeatability the worst feature. If a job tested on a sample of 100,000 fails on 500 million rows, it might run without a hitch. Therefore, local debugging becomes impossible, and the only way is to have extensive test environments.

4.4. Case Scenario 1: The Disappearing Rows

One of our streaming ETL systems found a recurring issue whereby the final output omitted a small fraction of enhanced user activity data. Data from Kafka came into the pipeline, upgraded using Hive reference data, and then was stored on S3. Weeks of research turned up a discrepancy in a Hive metastore schema. While a later schema modification created a new nullable column, the Spark job kept utilizing the same schema across joins. Spark gently eliminated data not matching the outmoded schema's join criteria; the change did not produce an error. Correcting the problem meant changing the Hive configuration and deleting cached metadata.

Lesson: When interacting with outside table systems, routinely check schema conformity. Spark's slowness in judgment could gently highlight variations.

4.4.1. Case Scenario 2: The Skewed Join That Stalled the Pipeline

One of the dubious problems that arose in a Spark project was merging historical clickstream data with product information. While the joint was permanently trapped in manufacturing, it operated flawlessly in development. Looking at the Spark UI, after forty minutes, two jobs were still under development while 198 of the 200 tasks had been completed. According to a thorough investigation, the two activities managed different keys; certain campaign IDs had over 10 million data while the rest had just a few thousand. We solved it by salting the hot keys and generating bespoke join logic to distribute the data using a skew join strategy. The running time fell from practically one hour to just twelve minutes.

Lesson: Look for variations in partition and junction width. Spark provides ambiguous information; use tools like df.rdd.glom() map(len) to examine partition imbalances.

5. Debugging Strategies That Work

Large-scale data system debuggingespecially those developed on Apache Sparkinvolves elements of science, intuition, and investigative investigation. Defects at scale could be difficult to identify, gently building under several layers of abstraction, and well disguised. Dealing with these challenges requires a mix of observability tools, Spark-specific insights, framework-agnostic techniques, and a modular and traceable attitude.

5.1. Framework-Agnostic Debugging Techniques

These techniques work well on most data systems and especially in cases when the basic reason is not clear-cut or when systems comprise several interconnected components.

5.1.1. Binary Search in Pipeline Stages

A divide-and-conquer strategy can save hours instead of meticulously looking over the whole pipeline. Separating the pipeline into several phasesdata intake, enrichment, joins, aggregation, and outputyou can carefully identify the error. Look at the intermediate outputs at every level; should the ultimate output be defective, then so should the others. Determine the beginning point where the data deviates and the last place the data seems to be true. This approach generally generates incorrect results or a change of viewpoint and limits the range of research.

5.1.2. Incremental Testing and Mocking

Doing large-scale projects takes time and money. Create other smaller test cases with representative piece production data. Model external systems like Hive or Kafka with local equivalents or static files. This enables, apart from cluster operation testing of transforms, solo. Mocking allows you to replicate edge events, such as null values, nonexistent fields, or schema incompatibilities, without affecting production data or settings.

5.1.3. Reconstructing DAGs in Notebooks

Rebuilding DAG (directed acyclic graph) logic in Jupyter or Databricks notebooks enhances interactive research for challenging projects. One can view schemas, make small adjustments, and check intermediate data frames. This helps you to better appreciate how your code creates execution plans and enhances troubleshooting. Every action usually reveals minute such things as mismatched schemas, incorrect join keys, or unanticipated null propagation by visual inspection.

5.2. Spark-Specific Debugging Techniques

Spark provides its own debugging tools and speed enhancements. Understanding them can help to prevent regressions and drastically lower resolution times.

5.2.1. Spark UI and Stage Analysis

Finding performance problems and stage faults calls for the Spark UI. It clarifies job lengths, memory utilization, execution guided acyclic graphs (DAGs), and shuffle operations.

Use it to:

- List ongoing or abandoned initiatives.
- Discover either too much data swapping or uneven partitions.
- Examine the executor's task distribution.
- Review logs for phases that fail.

This user interface enables you to convert raw data into insightful analysis, thereby enabling you to respond to questions like, Why is this phase slow? Which adjustment set off a shuffle? Are some keys generating more than required?

5.2.2. Broadcast Joins and Shuffle Operations

Join methods will guide the decision on the success or failure of Spark employment. Effective elimination of expensive shuffle joins is made possible in small dimension tables. Spark alone generates tables within a specified threshold; hand intervention usually produces superior results. One must be aware of how to apply broadcast joins differently from shuffle joins and identify instances where the optimizer makes bad decisions. This procedure tracks broadcast size, assesses fallback behavior, and controls thresholds.

5.2.3. Understanding Lineage and Caching Behavior

Data lineage is the road data follows during changes. Knowing lineage allows one to identify the causes of errors and clarifies surprising outcomes. First of significance is most certainly cache. Although caching can speed up iterative processes, inappropriate or overly extensive use may produce memory leaks or outdated results. Tracking the cached DataFrames, cases of their invalidation, and necessity for recalculation following further schema modifications is quite important. Using cached DataFrames between jobs without cleaning or renewing them is a typical error producing unequal results.

5.3. Tooling and Observability

Good debugging demands outstanding observability. Monitoring tools, log aggregators, and schedulers help to create the context and telemetry needed for a quick diagnosis.

5.3.1. Log Aggregation Tools

Distributed systems generate several logs. Combining logs and enabling filtering, pattern detection, and system correlation, Datadogthe ELK stackElasticsearch, Logstash, Kibanaallows:

.

For Spark, focus on:

- Emphasize out-of-memory problems, stack traces, task retirement, and Spark's executor logs.
- Driver notes on orchestral mistakes
- Logs of applications showing varying schema or configuration

Mark notes to increase searchability and traceability using task IDs, phases, and timestamps.

5.3.2. Profiling and Metrics Tools

Performance profiling tools such as Ganglia and Prometheus offer time-series metrics that show CPU, memory, disk, and network usage. They are the data sources that help you track down resource bottlenecks, the thrashing of executors, and the overhead of GC.

Profiling is particularly beneficial for:

- Finding places where memory is being lost or the stages showing slowdowns
- Analyzing resource saturation during the highest loads
- Matching job performance and infrastructure metrics

Also, reconfiguring metrics dashboards to send notifications after thresholds are exceeded can allow for issue resolution to be more proactive.

5.3.3. Airflow and Job Tracebacks

Airflow noms Spark jobs and aids in controlling dependencies and retries. Airflow's task logs and DAG visualizations are vital to grasping job sequences, failure spots, and retry behavior.

If a job fails:

- Leverage Airflow to locate the upstream dependencies
- Scan through task logs for any parameter mismatches or environment changes
- Verify that there are no inconsistent data partitions or that file references are not outdated

In addition, Airflow proves to be an excellent audit history repository, where the documents are describing when and how jobs are executed. Root cause analysis thus becomes more organized.

5.4. Mindset Shifts for Debugging at Scale

Debugging distributed systems requires a most fundamental change of viewpoint. Often inadequate are traditional, linear debugging methods. Successful engineers have to adopt specific cognitive models and adopt a holistic perspective:

- **One works between layers of abstraction:** Modern pipelines span multiple layers of reasoning: spark code, orchestration tools, storage systems, and metadata libraries. First one has to grasp the interactions among these parts. A problem can start in Spark but show up as an Airflow failure or a Hive schema difference. Good debugging calls for a fluid navigation of these layersknowledge of log locations, problem propagation, and tools available at every level.
- **Using clearly visible, modular code:** For large monolithic constructions, there are challenging diagnoses. Plan your pipelines with modular phases and well-defined constraints. Every level must be capable of independent re-execution, observable, and testable. Modular codes help to isolate errors and lower the blast radius during revisions. Use consistent measurements and orderly logging at every phase. This enables the linking of events across systems and helps to identify aberrant behavior.
- **Inducing traceability:** Traceability is the ability to connect inputs to outputs and changes to results. Distributed systems demand thoughtful versioning, schema management, and metadata monitoring. Among other tools, lineage trackers, Delta Lake, and Hive Metastore help to maintain this chain of custody. Traceability advances debugging, compliance, repeatability, and data pipeline confidence.

6. Scalable Design Principles from the Field

Building scalable data pipelines goes beyond basic code running. Growing systems manage guarantees of dependability, performance, maintainability, and adaptation. Having years experience negotiating real-world Spark pipelines and large data ecosystems, I have developed a set of design approaches that routinely generate strong and high-performance solutions. Fundamentally grounded on knowledge acquired via production difficulties, trial, error, and experimentation, these methods handle operational automation, pipeline logic, architecture, and performance optimization.

6.1. Data Pipeline Principles

6.1.1. Idempotency: The Cornerstone of Reliability

Idempotency basically means that a pipeline can be executed again safely without any change in the result. This is very important in distributed systems since the jobs may fail halfway and thus, they will be retried automatically. However, in the absence of idempotency, retries can cause the duplication of data, the increase of the metrics beyond the real value, or the changing of the downstream states to the wrong ones.

Some practical examples of idempotency are

- Saving outputs to divided paths according to the timestamp or business keys.
- Erasing or duplicating output paths before entering new results.
- Applying merge/upsert logic where possible (for instance, Delta Lake's MERGE INTO).

6.1.2. Checkpointing and Replayability

In large systems, mistakes are inevitable; checkpoints help to recover. Spark lets the system recover from the most recent successful state by enabling checkpointing for structured streaming. By use of intermediates in batch pipelines, manual checkpointing allows resuming without requiring all data from the start to be reprocessed. The preservation of unchangeable raw data calls for replayability. Running pipelines with past inputs will thus enable you to duplicate results should the transformation logic change.

6.1.3. Stateless vs Stateful Transformations

Transformations that do not carry state (like filters or simple mappings) are by nature very scalable and can be easily parallelized. Operations that carry state, however (for example, running totals, sessionization, and deduplication), are more

challenging to design correctly and need more attention in order not to bloat memory and to be correct.

Some of the best practices are

- Try to keep transformations as stateless as possible.
- When stateful logic is absolutely necessary, apply watermarks and windowed aggregations with time bounds in streaming pipelines.
- Save state in persistent storage if necessary (such as Redis or Delta Lake) instead of relying only on executor memory.

6.2. Codebase Design in Big Data Systems

6.2.1. Layered Architecture

Keep a clear distinction between various layers of your pipeline code:

- **Data Access Layer:** Manages input and output with external systems.
- **Transformation Layer:** Home of pure, testable business logic.
- **Orchestration Layer:** Manages jobs, dependencies, and retries.

This division not only turns the code into modular, but it also makes debugging, maintenance, and reuse much easier tasks.

6.2.2. Reuse and the DRY Principle

Data pipelines usually have similar logic in different domains. Avoid repeating by generalizing the common transformation patterns into shared utility functions or libraries.
.

For instance:

- Renaming or standardizing of columns
- Handling of null and changing of data type
- Normalization of time zones
- Validation of schema

The reusable logic not only better maintains but also reduces the incidence of bugs, especially when new data sources are introduced or business rules are changed.

6.2.3. Testable Pipeline Designs

Unit testing Spark logic is perfectly possibleand, in fact, necessary. Concentrate on the transformation layer, where non-random logic can be verified with sample data.

- **Unit tests:** Confirm specific changes (column derivatives, filtering logic) by employing very small DataFrame simulations.
- **Integration tests:** Measure end-to-end power over different stages with standard datasets and temporary storage.

By simulating the outside dependencies, pipelines can be tested independently for S3, Hive, and Kafka. Each improvement, however, must be supported by tests that are integrated into the CI process.

6.3. Performance Tips

6.3.1. Partition Tuning

Usually, default partitioning results in inefficiencieseither an excess of unimportant jobs (which results in expense) or a shortage of critical ones (which generates data skew). Spark permits the repartition () and coalesce () adjustment of partition counts. With repartition (n), improve parallelism and increase partitions.

- Use coalesce(n) to reduce partitions when writing so as to stop the development of too few output files.
- See Spark UI's partition sizes; optimum throughput requires each job to run between 100 and 200 MB. Pursue balance.

6.3.2. Avoiding Wide Transformations

Operations such as groupByKey (), distinct(), and wide join() result in costly shuffles. These operations transfer data across the cluster and are the main source of latency and failure.

Mitigation strategies include:

- Substituting groupByKey() with reduceByKey() or aggregateByKey()
- Opting for broadcast joins if one side is small
- Doing pre-aggregations to limit the amount of data being processed in the wide operations

6.3.3. Minimizing Shuffles

Shuffles are expensive because they involve disk and network I/O. Some methods to lessen or even get rid of them are

- Keep in memory intermediate results that are used again in subsequent stages.
- Pre-sort data if the logic of the downstream requires ordered data.
- Employ partitioning schemes that match join keys or output locations.

By utilizing explain () to examine the physical plan, it is possible to discover unintentional shuffles.

6.4. CI/CD for Data Pipelines

Adopting DevOps methodology in data engineering allows one to get the same results, efficiently, and faster in building and launching pipelines.

6.4.1. Automated Tests and Linters

Using version control repositories for data pipeline logic. Implementing automated testing frameworks (for example, using PyTest for PySpark) to confirm during each commit of code that the transformations are correct. Linters and formatters give the guarantee of code conformity in teams.

Before merging changes, the following must be done:

- Execute all the unit and integration tests
- Ensure that the schema is compatible
- Highlight the modifications in the transformations that are no longer in use or in the configuration that has been changed

6.4.2. Deployment Automation

Using Airflow, Databricks tasks, and CI/CD pipelinesthat is, GitHub Actions, Jenkins, and GitLab CIcreate, bundle, and deploy Spark tasks to your orchestrator. Provide dependent packaging, Spark configuration validation, environmental impact, and environment automation.

6.4.3. Canary Runs and Data Validation Guards

Run canary activities on a smaller data set before rolling pipeline changes globally. Review measures in line with historical standards covering row counts, null rates, and value distributions.

Check data quality at key points.

- **Schema validation:** Verify if the column types and the number of columns are the same as what you have in your mind.
- **Volume checks:** Monitor the volume by observing row counts and distribution changes.
- **Value checks:** Highlight certain empty cells, repeated cells, or wrong value verification cells.

These shields facilitate spotting bugs early and increase pipeline output confidence.

7. Case Study: Building and Operating a Real-Time Analytics Pipeline

From a luxury in the new digital terrain, real-time analytics has become a basic need. Businesses have to react fast to customer behavior, see problems straight away, and provide operational openness with live dashboards. Using Apache Kafka, Spark Structured Streaming, and Cassandra a real-time customer event processing pipeline's design, implementation, and operation are described in this case study. Along with the methods applied, it also examines the challenging issues encountered: backpressure control, precisely-once processing, schema evolution, and diagnosis of enigmatic production problems.

7.1. Background: Business Need for Real-Time Insights

On an expanding e-commerce platform, the obvious but technically challenging business objective was enabling real-time tracking and analytics of user activity. Marketing teams wanted real-time access to under five-second latency product clicks, cart adds, and transactions. The objective was to retain data quality and completeness while filling a live dashboard with real-time activity, even under unexpected traffic spikes. batch processing turned out to be not enough. Even ten to fifteen minutes delay opportunities for upselling tactics, fraud detection, and customer service interventions. The team opted to build a real-time data pipeline guaranteed to be robust in absorbing, processing, and distributing millions of events every hour.

7.2. Architecture Overview

To achieve corporate objectives, we've established a streaming pipeline that consists of the following components:

- **Kafka:** Kafka was the ingestion layer and gathered consumer events from many microservices and browsers via REST and WebSocket APIs, and thus, it was the ingestion layer for Kafka. Topics were set in accordance with the type, that is, product view, add to a basket, and checkout.
- **Spark Structured Streaming:** Continuous data processing was supported by Spark Structured Streaming. The streaming task took over raw JSON messages and then did the aggregations that it improved by metadata, i.e., session data and customer attributes.
- **Cassandra:** Cassandra was selected as the serving layer. Due to low-latency read and write operations, the serving layer was easily accessible by using time-partitioned tables, thus being able to store the processed events for the dashboards as well as for further microservice consumption.
- **Dashboard Layer:** The React-based user interface gets near-real-time data from Cassandra; thus, it can visualize important performance metrics such as user sessions, conversion funnels, and top products over certain time periods.

The pipeline was designed on Kubernetes, and Kafka consumers, along with Spark executors, were autoscaled to handle traffic spikes.

7.3. Complexities in the Real World

7.3.1. Handling Backpressure

Traffic was pretty up and down, reaching its highest point around product launches or similar events. We had to ensure that the system was capable of handling not only slow traffic but also sudden bursts without breaking down. Even though we had to be careful while changing the batch interval and the maximum offset per trigger, Spark Structured Streaming's backpressure feature was quite useful. Unfit settings could either limit the system too much and increase the delay or change it too much and cause memory shortages. At last, we decided to use adaptive batching which is based on the current latency and the system condition.

7.3.2. Exactly-Once Semantics

In a streaming setting, precisely-once delivery semantics is notably challenging. Retries could cause events to be replayed, copied in Kafka as a result of producer retries, or partially handled depending on job restarts.

To enforce exactly-once guarantees:

- Kafka offsets exactly matched successful Cassandra writings.
- Timestamps and event IDs let the deduplication system avoid consecutive writes.
- To check previously handled data, an idempotency layer was set up during the write phase.

Although data integrity was quite important, this required storing a light weight processing state and tolerating some write overhead.

7.3.3. Schema Evolution

As the event model evolved, new fieldse.g., referrer_url, coupon_codewere added even while existing data changed their format. Dynamic schema parsing and version-sensitive enrichment methods are needed for control of these modifications free from interruption. Event generators tagged payloads with schema versions applied a schema registry technique. The Spark project applied appropriate deserializing and transformation logic using these tags. Unknown domains were forwarded for investigation to a dead-letter queue (DLQ).

7.4. Debugging in Production: A Case of Lag and Dropped Events

The team saw occasional slowdown and missing dashboard data following many months of perfect performance. Tests of spark latency revealed periodic spikes; some types of events suggested Cassandra writes interrupted patterns. The uneven character of the problem makes replication in smaller settings challenging.

7.4.1. Initial Observations:

- Kafka consumer lag remained high for certain partitions.
- Spark stages appeared to be processing old data instead of catching up.
- Cassandra writes showed fewer rows than expected compared to raw Kafka volume.

7.4.2. Root Cause Analysis:

In our investigation of Spark logs, UIs, and config items, we discovered that the root cause was a combination of two misconfigurations:

- **Watermark Misconfiguration:** A fixed watermark of ten minutes was agreed upon to eliminate late events. However, due to network congestion, events were delayed not only beyond this threshold but also at the peak demand times, and without the knowledge of the users, they were removed from the delayed Spark dataset.
- **Improper Checkpointing:** The checkpoint position from the last redeployment is the source of residual state instability. As a result, Spark reprocesses a part of the previous batches with restarts, thus elongating the processing time and delaying the intake of new data.

These issues are each complicated on their own but when taken together, they conceal the actual source of the problems. Watermarking issues resulted in data loss, checkpointing induced latency and decreased processing speed.

7.4.3. Resolution Steps:

- Increased the watermark buffer to 30 minutes, with monitoring on late event rates.
- Cleaned and re-initialized the checkpoint location during a controlled downtime.
- Added structured logging for late events and dead-letter records.
- Introduced metrics to alert on lag duration, partition staleness, and output row counts.

7.5. Results and Key Learnings

Subsequently, with a major fix, the pipeline became much more stable. The Kafka lag went back to almost zero, and the data completeness improved by more than 99%. The dashboard latency went from an average of 15 seconds to nearly 5 seconds, also under peak loads.

Key Learnings:

- **Backpressure handling is a balancing act:** The choice of the batch interval and the offset may hurt more than help. Essential are the observability and the adaptive configurations.
- **Watermarks need the never-ending adjustment:** A definite period is fine for today but could be wrong tomorrow if the traffic conditions change. Let them be flexible and observable.
- **Checkpointing is a delicate but primary part:** If there is any corruption in the state or it becomes inconsistent, errors like duplication, staleness, or data loss can occur. The clean deployment processes and the checks of state validation are so important.
- **Visibility is your friend:** In case of the loss, delay, or refusal of an event, do not forget to add logs in such a case. An idea of how many events are in each step would be great. Without the visibility, the debugging process can become like that of working from guesses.
- **Design for failure, not perfection.** By using dead-letter queues, retries, and idempotent writes, the system can absorb transient and silent faults without a glitch.

8. Conclusion and Reflections

From technical and professional perspectives, the change from SQL-based systems to Apache Spark-powered architectures has been transformational. SQL brought a slow

learning curve, simplicity, and clarity. It gave the ideal basis for orderly data storage and batch processing. Its boundaries were obvious, though, as corporate demands for real-time insights expanded along with data volumes. Spark allowed scalability and speed unachievable with SQL alone by means of its distributed computing approach. Still, extra capacity brought complexity: observability became crucial, debugging grew diverse, and design decisions had increasingly substantial repercussions at scale. Based on what I have observed, not just technicalities but also intuition and pattern identification along with logs and stack traces, define debugging of major data systems. Defining data lineage throughout numerous phases, recognizing erroneous transformations, or optimizing operations to fit the architecture instead of against it has a subtle capacity to help with architectural fit. But it's also a science with exact diagnostic tools, rigorous techniques, and reproducible foundations. The basic concept is to treat systems holistically, viewing them not simply as chores or scripts but as linked processes demanding resilience, observability, and deliberate design.

For those beginning this road, my counsel is to welcome the shift with caution and inquiry. Not only the APIs; also know the fundamental concepts. Experiment with parts all through development to learn their manufacturing value without delay. Discover also the holistic view of systems. Debugging involves not only seeing the poor line of code but also knowledge of the linkages across inputs, infrastructure, transformations, and outputsoften in surprising directions. Researching new paradigms with improved scalability and developer usability interests me. Native Kubernetes spark implementations provide elastic computing and enhanced resource orchestration. Delta Lake brings ACID transactions and aggregates batch-streaming capability into data lakes. Real-time feature stores are linking production-level streaming systems into machine learning pipelines. The process goes beyond basic tool knowledge to include shifting our view of data systems in an environment where dependability, scalability, and speed are critical. Every failure and final resolution helps us to increase our capacity as engineersnot only of code but also of systems affecting next decisions.

References

[1] Armbrust, Michael, et al. "Spark sql: Relational data processing in spark." *Proceedings of the 2015 ACM SIGMOD international conference on management of data*. 2015.

[2] Syed, Ali Asghar Mehdi, and Erik Anazagasty. "AI-Driven Infrastructure Automation: Leveraging AI and ML for Self-Healing and Auto-Scaling Cloud Environments." *International Journal of Artificial Intelligence, Data Science, and Machine Learning* 5.1 (2024): 32-43.

[3] Gulzar, Muhammad Ali, et al. "Bigdebug: Debugging primitives for interactive big data processing in spark." *Proceedings of the 38th International Conference on Software Engineering*. 2016.

[4] Kumar Tarra, Vasanta, and Arun Kumar Mittapelly. "AI-Driven Lead Scoring in Salesforce: Using Machine Learning Models to Prioritize High-Value Leads and Optimize Conversion Rates". *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 5, no. 2, June 2024, pp. 63-72

[5] Guller, Mohammed. "Big data analytics with spark." *ISBN-13 (pbk)* (2015): 978-1.

[6] Jani, Parth, and Sangeeta Anand. "Compliance-Aware AI Adjudication Using LLMs in Claims Engines (Delta Lake + LangChain)". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 2, May 2024, pp. 37-46

[7] Karau, Holden, and Rachel Warren. *High performance Spark: best practices for scaling and optimizing Apache Spark*. " O'Reilly Media, Inc.", 2017.

[8] Chaganti, Krishna Chaitanya. "AI-Powered Threat Detection: Enhancing Cybersecurity with Machine Learning." *International Journal of Science And Engineering* 9 (2023): 10-18.

[9] Talakola, Swetha. "The Optimization of Software Testing Efficiency and Effectiveness Using AI Techniques". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 5, no. 3, Oct. 2024, pp. 23-34

[10] Marra, Matteo. *A live debugging approach for big data processing applications*. Diss. Ph. D. thesis, Vrije Universiteit Brussel, 2022.

[11] Lalith Sriram Datla, and Samardh Sai Malay. "Transforming Healthcare Cloud Governance: A Blueprint for Intelligent IAM and Automated Compliance". *Journal of Artificial Intelligence & Machine Learning Studies*, vol. 9, Jan. 2025, pp. 15-37

[12] Tang, Shanjiang, et al. "A survey on spark ecosystem: Big data processing infrastructure, machine learning, and applications." *IEEE Transactions on Knowledge and Data Engineering* 34.1 (2020): 71-91.

[13] Arugula, Balkishan. "Prompt Engineering for LLMs: Real-World Applications in Banking and Ecommerce". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Jan. 2025, pp. 115-23

[14] Gulzar, Muhammad Ali. *Automated testing and debugging for big data analytics*. University of California, Los Angeles, 2020.

[15] Abdul Jabbar Mohammad, and Guru Modugu. "Behavioral TimekeepingUsing Behavioral Analytics to Predict Time Fraud and Attendance Irregularities". *Artificial Intelligence, Machine Learning, and Autonomous Systems*, vol. 9, Jan. 2025, pp. 68-95

[16] Damus Ros, Nicolas. "A Business Intelligence Solution, based on a Big Data Architecture, for processing and analyzing the World Bank data." (2023).

[17] Veluru, Sai Prasad. "Bidirectional Curriculum Learning: Decelerating and Re-Accelerating Learning for Robust Convergence". *International Journal of Emerging Trends in Computer Science and Information Technology*, vol. 5, no. 2, June 2024, pp. 93-102

[18] Jambi, Sahar Hussain. *Engineering Scalable Distributed Services for Real-Time Big Data Analytics*. Diss. University of Colorado at Boulder, 2016.

[19] Chaganti, Krishna Chaitanya. "Securing Enterprise Java Applications: A Comprehensive Approach." *International Journal of Science And Engineering* 10.2 (2024): 18-27.

[20] Bhaskaran, Shinoy Vengaramkode. "Integrating data quality services (dqs) in big data ecosystems: Challenges, best practices, and opportunities for decision-making." *Journal of Applied Big Data Analytics, Decision-Making, and Predictive Modelling Systems* 4.11 (2020): 1-12.

[21] Al Samisti, Fanti Machmount. "Visual Debugging of Dataflow Systems." (2017).

[22] Balkishan Arugula, and Suni Karimilla. "Modernizing Core Banking Systems: Leveraging AI and Microservices for Legacy Transformation". *Artificial Intelligence, Machine Learning, and Autonomous Systems*, vol. 9, Feb. 2025, pp. 36-67

[23] Zhang, Jian. "Exploring and Evaluating the Scalability and Efficiency of Apache Spark using Educational Datasets." (2018).

[24] Allam, Hitesh. "Intent-Based Infrastructure: Moving BeyondIaC to Self-Describing Systems". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Jan. 2025, pp. 124-36

[25] Talakola, Swetha. "Transforming BOL Images into Structured Data Using AI". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Mar. 2025, pp. 105-14

[26] Akil, Bilal. *A Comparative Study of Hadoop MapReduce, Apache Spark & Apache Flink for Data Science*. Diss. 2018.

[27] Jabbar Mohammad, Abdul. "Integrating Timekeeping and Payroll Systems During Organizational TransitionsMergers, Layoffs, Spinoffs, and Relocations". *Los Angeles Journal of Intelligent Systems and Pattern Recognition*, vol. 5, Feb. 2025, pp. 25-53

[28] Veluru, Sai Prasad, and Mohan Krishna Manchala. "Using LLMs as Incident Prevention Copilots in Cloud Infrastructure." *International Journal of AI, BigData, Computational and Management Studies* 5.4 (2024): 51-60.

[29] Jani, Parth. "Modernizing Claims Adjudication Systems with NoSQL and Apache Hive in Medicaid Expansion Programs." *JOURNAL OF RECENT TRENDS IN COMPUTER SCIENCE AND ENGINEERING (JRTCSE)* 7.1 (2019): 105-121.

[30] Wolohan, John. *Mastering Large Datasets with Python: Parallelize and Distribute Your Python Code*. Simon and Schuster, 2020.

[31] Datla, Lalith Sriram. "Infrastructure That Scales Itself: How We Used DevOps to Support Rapid Growth in Insurance Products for Schools and Hospitals". *International Journal of AI, BigData, Computational and Management Studies*, vol. 3, no. 1, Mar. 2022, pp. 56-65

[32] Kupunarapu, Sujith Kumar. "Data Fusion and Real-Time Analytics: Elevating Signal Integrity and Rail System Resilience." *International Journal of Science And Engineering* 9 (2023): 53-61.

[33] Allam, Hitesh. "Code Meets Intelligence: AI-Augmented CI CD Systems for DevOps at Scale." *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 6, no. 1, Jan. 2025, pp. 137-46

[34] Sangaraju, Varun Varma, and Senthilkumar Rajagopal. "Applications of Computational Models in OCD." *Nutrition and Obsessive-Compulsive Disorder*. CRC Press, 2023. 26-35.

[35] Bagherzadeh, Mehdi, and Raffi Khatchadourian. "Going big: a large-scale study on what big data developers ask." *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 2019.

[36] S. S. Nair, G. Lakshmikanthan, J.ParthaSarathy, D. P. S, K. Shanmugakani and B.Jegajothi, ""Enhancing Cloud Security with Machine Learning: Tackling Data Breaches and Insider Threats,"" 2025 International Conference on Electronics and Renewable Systems (ICEARS), Tuticorin, India, 2025, pp. 912-917, doi: 10.1109/ICEARS64219.2025.10940401.

[37] R. Daruvuri, K. K. Patibandla, and P. Mannem, "Data Driven Retail Price Optimization Using XGBoost and Predictive Modeling", in *Proc. 2025 International Conference on Intelligent Computing and Control Systems (ICICCS), Chennai, India. 2025*, pp. 838–843.