JETIR.ORG ISSN: 2349-5162 | ESTD Year : 2014 | Monthly Issue
JOURNAL OF EMERGING TECHNOLOGIES AND INNOVATIVE RESEARCH (JETIR)
An International Scholarly Open Access, Peer-reviewed, Refereed Journal

Cost Optimization in Data Engineering

Dr. B. Rajesh Kumar
Associate Professor
Department of Software Systems
Sri Krishna Arts and Science College,
Coimbatore, India
rajeshkumarb@skasc.ac.in

Praveen J
Student
Department of Software Systems
Sri Krishna Arts and Science College,
Coimbatore, India
praveenj23mss036@skasc.ac.in

ABSTRACT

Cloud-based Spark data pipelines in product-based companies process large volumes of financial and customer data on Amazon EMR, and inefficiencies in storage layout, executor configuration, and pre-production validation translate directly into recurring infrastructure cost. This paper presents a practical, config-first approach to cost optimization for data engineering pipelines, built around three complementary techniques: S3 small-file compaction through partition-aware REPARTITION hints, Spark job tuning through executor sizing, Adaptive Query Execution (AQE), and dynamic allocation, and pre-production verification of every change through an automated EMR trigger-and-test workflow before the change reaches production. Unlike approaches that treat cost reduction as a one-time infrastructure exercise, this paper proposes an integrated optimization lifecycle in which storage layout, compute configuration, and pre-production testing are addressed together, in a fixed order that always prefers non-destructive, output-preserving changes over logic or data changes. Observations from applying this approach to real ETL jobs show cost reductions of up to 90% in vCPU-hour and GB-hour consumption for individual jobs, with automated pre-production validation confirming that optimized jobs preserve identical output row counts. The findings indicate that most Spark cost overruns in production pipelines are attributable to configuration and storage-layout drift rather than application logic, and that systematic, tool-assisted analysis can recover most of this cost without code changes.

Keywords—Cost Optimization, Apache Spark, Amazon EMR, S3 Small-File Compaction, Spark Tuning, Adaptive Query Execution, Dynamic Allocation, Automated Preproduction Testing Framework, Pre-Production Testing, Cloud Data Engineering

I. INTRODUCTION

Modern financial-technology platforms increasingly rely on large-scale Spark pipelines running on Amazon EMR to process transactional, customer, and analytics data. In many such organizations, the data engineering team operates a large fleet of ETL Spark jobs orchestrated through a batch processing platform and a cluster configuration layer, executing on EMR Serverless and EMR EC2. As the number of pipelines and the volume of data they process has grown, so has the cloud compute bill associated with running them.

Three recurring sources of avoidable cost were observed across these pipelines. First, tables accumulate a large number of small files in Amazon S3 over time, caused by frequent, low-volume writes; this increases the number of file-open and list operations Spark must perform and degrades scan efficiency. Second, Spark jobs are frequently configured with executor and memory settings that no longer match the actual data volume or shape of the job, resulting in idle compute capacity, excessive shuffle partitions, or executor spill. Third, changes intended to fix either of the above are often deployed directly to production without a reliable pre-production verification step, creating risk that a cost-motivated change silently alters job output.

This paper describes a combined approach that addresses all three issues as a single optimization lifecycle rather than as isolated fixes: S3 compaction to reduce small-file overhead, Spark configuration tuning to right-size compute resource usage, and automated EMR trigger testing to validate every change against a real EMR run in a preproduction environment before it is promoted to production.

II. BACKGROUND AND MOTIVATION

Apache Spark's performance on cloud object storage such as Amazon S3 is highly sensitive to file layout. A large number of small files increases the overhead of file listing and task scheduling relative to the actual volume of data processed, a problem widely referred to as the 'small-file problem.' Compaction — merging many small files into fewer, appropriately sized files — is a well-established mitigation, but it must be applied carefully, since it modifies stored data and can affect downstream jobs if the resulting partition count is chosen incorrectly.

Independently, Spark job cost on EMR is a direct function of allocated executor resources, the duration for which they are held, and how efficiently they are used during that time. Idle executors awaiting shuffle data, oversized executor memory that is never used, and shuffle partition counts mismatched to data volume are common and often invisible sources of waste that do not require any change to business logic to fix.

Finally, any change made purely to reduce cost — whether a REPARTITION hint or an executor-sizing change — carries a risk of unintentionally altering the job's output if the change is not verified. Pre-production frameworks that can trigger a real run against representative data and confirm output equivalence before a change reaches production close this verification gap.

III. LITERATURE SURVEY

Existing research on Spark and cloud cost efficiency has generally treated storage optimization and compute optimization as separate problems. Armbrust et al. [1] describe the architectural basis of Spark SQL's Catalyst optimizer and the role of the Adaptive Query Execution (AQE) framework in dynamically adjusting shuffle partitions and join strategies at runtime, which forms the basis of several compute-side tuning techniques used in this paper.

Zaharia et al. [2] characterize the cost and performance trade-offs of in-memory cluster computing frameworks, noting that resource under-utilization, rather than raw compute capacity, is the dominant driver of inefficiency in long-running production clusters. Armbrust et al. [3] further discuss the operational challenges of running Spark reliably at scale, including configuration drift between environments, which motivates the pre-production verification approach adopted in this paper.

On the storage side, prior work on the Hadoop Distributed File System and cloud object stores [4] has documented the small-file problem and established compaction as a standard remedial technique, while cautioning that compaction is a data-mutating operation that requires careful partition sizing and downstream sign-off. Amazon Web Services' own EMR Serverless and pricing documentation [5] provides the vCPU-hour and GB-hour cost model used for quantifying savings in this paper.

Collectively, this literature supports the view taken in this paper: that cost optimization is most effective and lowest-risk when compute tuning, storage-layout correction, and pre-deployment verification are treated as a single, ordered lifecycle rather than as independent interventions.

IV. S3 SMALL-FILE COMPACTION

The first component of the optimization lifecycle addresses the small-file problem in Amazon S3-backed Hive tables. Because production tables cannot be inspected directly by listing S3 objects at scale, the workflow instead determines table size and partitioning structure using the Statistics section of a Hive DESCRIBE FORMATTED query, which reports total table size and row count without requiring direct S3 access.

Using the reported total size and the target file size for efficient Spark scanning (approximately 128 MB per output file, matching Spark's default read-split size), an appropriate REPARTITION value is calculated as total table size divided by target file size. This value is added as a REPARTITION hint directly in the affected ETL configuration file, so that the job's write stage consolidates output into the calculated number of files instead of the many small files produced by frequent low-volume writes.

Because this change modifies the physical layout of stored data, it is treated as a flagged, sign-off-required change rather than a routine configuration edit: a JIRA ticket documenting the affected table, the calculated repartition value, and the justification is raised, and a pull request against the ETL pipeline configuration repository is opened for review before merge. This ensures that compaction, unlike pure compute tuning, is never applied silently.

V. SPARK JOB TUNING

The second and most frequently applied component of the lifecycle is configuration-only Spark tuning, applied strictly in the following order, with the process stopping as soon as a job meets its cost target so that no unnecessary risk is introduced: executor and memory sizing, shuffle and partition tuning, dynamic allocation and idle-capacity tuning,

and file/IO tuning. Only if these four exhaust their potential is a minimal, output-preserving code change considered, and only with explicit sign-off.

Execute and memory sizing right-sizes spark.executor.cores, spark.executor.memory,spark.executor.memoryOverhead, and spark.executor.instances to the actual workload, preferring dynamic allocation over a fixed executor count; for EMR Serverless, worker vCPU is constrained to the values 1, 2, 4, 8, 16, or 32 with correspondingly bounded memory tiers. Shuffle and partition tuning enables Adaptive Query Execution(spark.sql.adaptive.enabled,coalescePartitions, and skewJoin), targets approximately 128 MB per shuffle partition, and raises the broadcast join threshold where applicable to avoid unnecessary shuffles . File tuning applies coalesce or repartition on write to avoid producing new small files, and confirms Parquet with Snappy compression and partition pruning are in effect.

The financial impact of each proposed change is quantified using an EMR Serverless cost model of vCPU-hours multiplied by a per-vCPU-hour rate plus GB-hours multiplied by a per-GB-hour rate (with an analogous instance-hour plus EBS model for EMR EC2), and reported as a structured Before-versus-After comparison covering wall-clock duration, average worker count, vCPU-hours, GB-hours, estimated cost per run, and percentage cost decrease. In one representative job observed under this workflow, right-sizing executor and shuffle configuration reduced wall-clock time from approximately 42 minutes to 18 minutes, worker count from 120 to 18, and estimated cost per run by approximately 90%, without any change to job output.

VI. AUTOMATED EMR TRIGGER TESTING IN PREPRODUCTION

The third component of the lifecycle addresses verification: every compaction or tuning change is validated against a real EMR execution in a preproduction environment before it is promoted to production, using an automated preproduction testing framework. Given an ETL pull request, the associated changed configuration file and Jira ticket are first identified, and the underlying pipeline name and runtime engine are resolved by querying pipeline metadata and parsing the processor's runtime arguments (compute type, environment, EMR group, project, and branch).

These resolved values are assembled into a trigger payload and submitted to the processing API to start a job run on EMR in the automation preprod environment. Because triggering a real EMR run is a consequential, resource-consuming action, this step is gated on explicit confirmation before submission. Once triggered, run progress is tracked across four sequential modules — Extract, Generate, Execute, and Test — primarily by polling the automation bot's status message, since the corresponding REST status endpoint is often unreliable mid-run.

The Execute module is of particular importance for verifying compaction and tuning changes: it reports a Before versus After row-count comparison for every target table in a transient schema created specifically for the test. For a pure repartition, compaction, or resource-configuration change, the expected row-count delta between before and after is zero, providing direct evidence that the optimization preserved output while changing only physical layout or resource consumption.

On a passing result, the run is recorded as verified and the estimated cost savings computed during Spark tuning are upgraded from an estimate to a measured figure. On a failing result, the workflow distinguishes platform-level failures from job-configuration failures, retrieves failure logs from the automation bot and the pipeline debugging tool, searches prior resolutions in the platform-support channel, and — for job-configuration failures only — applies a minimal corrective change to the configuration file and raises a follow-up pull request for human review; fixes are never merged automatically.

VII. PROPOSED INTEGRATED COST OPTIMIZATION LIFECYCLE

This paper proposes an integrated lifecycle that sequences the three techniques above so that the least risky, most easily reversible changes are always attempted first. The lifecycle begins with Spark configuration analysis using a monitoring tool against the Spark History Server; if executor sizing, shuffle tuning, or dynamic allocation adjustments alone meet the cost target, no further change is required. If small-file overhead is identified as a contributing factor, a REPARTITION hint is calculated from Hive statistics and proposed as a flagged, sign-off-gated change. Only if configuration and storage-layout changes are insufficient is a minimal, output-preserving code change considered, and only with explicit user agreement.

Every change produced by this lifecycle, regardless of type, is passed through the automated EMR trigger-and-test step before being deployed to production, so that a Before-versus-After cost comparison is always paired with a Before-versus-

After correctness comparison. This ensures that cost reduction and output correctness are verified together rather than as separate, sequential concerns.

VIII. RESULTS AND OBSERVATIONS

Applying this lifecycle to representative ETL jobs within the pipeline fleet produced consistent qualitative and quantitative observations. Jobs with a high small-file count on their source or intermediate tables showed measurable scan-time reduction once a calculated REPARTITION hint was applied, with the Execute-module row-count check confirming zero row-count drift after compaction.

Jobs tuned purely at the configuration level — executor sizing, AQE, and dynamic allocation — showed the largest and most consistent cost reductions, in the observed representative case reducing estimated per-run cost by approximately 90% (wall-clock time reduced from about 42 to 18 minutes, average worker count reduced from 120 to 18) without any code or data change. Jobs that reached the code-change escalation path were rare, occurring only when configuration and storage-layout tuning had already been exhausted, consistent with the lifecycle's design intent of treating logic changes as a last resort.

Across all cases, automated pre-production testing before deployment caught configuration errors that would otherwise have surfaced only in production, and the Before/After row-count check embedded in the Execute module provided a repeatable, automated way to confirm that a cost-motivated change had not altered job output.

IX. DISCUSSION

These results indicate that a substantial share of cloud cost in production Spark pipelines is recoverable through configuration and storage-layout correction alone, without touching application logic or stored data semantics. This has two practical implications for data engineering teams operating at scale: first, cost optimization efforts should default to the lowest-risk lever available and escalate only when necessary, rather than defaulting to code changes; second, the availability of low-friction pre-production verification, such as an automated trigger-and-test workflow, is what makes it safe to apply compaction and tuning changes with confidence, since it converts an estimated cost saving into a measured, output-verified one before the change reaches production.

The approach also illustrates that cost and correctness need not be treated as competing concerns: by gating every optimization behind an automated row-count equivalence check, the lifecycle allows aggressive cost tuning without a corresponding increase in production risk.

X. LIMITATIONS OF THE STUDY

The cost figures reported in this paper are drawn from representative jobs observed during the application of this workflow and are not the result of a controlled experimental study across the full pipeline fleet; results will vary by job shape, data skew, and cluster type (EMR Serverless versus EMR EC2). The REPARTITION hint calculation depends on the accuracy of Hive table statistics, which may be stale if a table has not been recently analyzed. The automated verification step depends on the availability and reliability of the testing backend and its chat-based status reporting, which was observed to occasionally require fallback to secondary status signals.

XI. CONCLUSION

This paper presented an integrated, config-first approach to cloud data pipeline cost optimization combining S3 small-file compaction, Spark configuration tuning, and automated EMR trigger testing in preproduction. By ordering these techniques from least to most invasive and by verifying every change against a real EMR run before production deployment, the approach recovers significant cost — up to approximately 90% in the representative case studied — while maintaining output correctness. The results suggest that most Spark cost overruns in production pipelines originate in configuration and storage-layout drift rather than in application logic, and that systematic, tool-assisted analysis paired with automated pre-production verification can address this without introducing correctness risk.

XII. FUTURE SCOPE

Future work can extend this lifecycle with automated, continuous drift detection that proactively flags jobs whose configuration or file layout has drifted from best practice, rather than relying on ad hoc analysis. Further integration between the cost model and the automated verification step could allow measured (post-verification) cost savings to be tracked automatically over time across the full pipeline fleet, and machine-learning-based prediction of optimal executor and partition configuration from historical Spark History Server data is a promising direction for reducing the manual analysis effort described in this paper.

REFERENCES

[1] M. Armbrust, R. S. Xin, C. Lian et al., "Spark SQL: Relational Data Processing in Spark," in Proc. ACM SIGMOD Int. Conf. Management of Data, 2015, pp. 1383–1394.

[2] M. Zaharia, M. Chowdhury, T. Das et al., "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in Proc. USENIX NSDI, 2012.

[3] M. Armbrust, T. Das, L. Sun et al., "Delta Lake: High-Performance ACID Table Storage over Cloud Object Stores," Proc. VLDB Endowment, vol. 13, no. 12, pp. 3411–3424, 2020.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in Proc. IEEE 26th Symp. Mass Storage Systems and Technologies (MSST), 2010, pp. 1–10.

[5] Amazon Web Services, "Amazon EMR Serverless Pricing and Best Practices," AWS Documentation, 2023.