# Horizontal vs. Vertical Executor Scaling for TPC-DS Analytical Workloads on Geo-Distributed Kubernetes

**Italo V. P. Guimaraes**[1], **Aleteia Araujo**[1]

[1]Computer Science Department (CIC), University of Brasília (UnB)
Brasília – DF – Brazil

italo.guimaraes@aluno.unb.br, aleteia@unb.br

***Abstract.*** *When deploying Apache Spark on Kubernetes, practitioners must decide whether to allocate resources as many small executors or as few large ones. The conventional argument for large executors - that they reduce shuffle redistribution - rests on an assumption that Adaptive Query Execution (AQE) in Spark 3.5 renders false: shuffle volume is virtually identical across all executor sizes, because AQE adapts partition plans to the data at runtime, not to executor configuration. Without a shuffle advantage, vertical scaling only adds JVM garbage-collection overhead: large executors accumulate up to 84% more GC time, and horizontal scaling delivers 22% shorter execution at the same resource budget. Practitioners should prefer small executors (4 cores, 14 GB) and scale out.*

## 1. Introduction

Distributed analytical processing underlies a growing fraction of organizational workloads, from business intelligence dashboards to large-scale data science pipelines. Apache Spark [Zaharia et al. 2016] on Kubernetes [Burns et al. 2016] has emerged as a dominant execution substrate, and Delta Lake [Armbrust et al. 2020] provides transactional guarantees over object storage in this setting.

When sizing Spark deployments, practitioners frequently face a fundamental trade-off between allocating resources as *many small executors* (horizontal scaling) or as *few large executors* (vertical scaling). Conventional wisdom generally favors larger executors for shuffle-intensive workloads, based on the assumption that a reduced number of executors decreases the amount of shuffle partitions and minimizes inter-node data transfers [Zaharia et al. 2016]. However, this assumption was established prior to the introduction of Adaptive Query Execution (AQE) [Zhu et al. 2020] in Spark 3.0. AQE dynamically optimizes execution plans at runtime according to observed data statistics, including the coalescing and reorganization of shuffle partitions. Consequently, the traditional relationship between executor sizing and shuffle efficiency may no longer hold in modern Spark deployments.

Despite the widespread adoption of AQE, its impact on the effectiveness of horizontal versus vertical executor scaling for large-scale analytical workloads executed on real distributed infrastructures remains insufficiently understood. Previous experiments conducted on the same infrastructure indicated that configurations based on Small executors achieved near-linear horizontal scalability at SF500, while exhibiting lower efficiency at SF100. These findings raise an important open question: whether configurations using

larger executors are capable of mitigating these inefficiencies and improving performance at smaller scale factors.

This paper extends that line of inquiry by fixing the scale factor and varying executor size. We compare three executor configurations: Small (4 cores, 14GB), Medium (8 cores, 28GB), and Large (12 cores, 48GB) across SF100, SF250, and SF500 using TPC-DS (Transaction Processing Performance Council - Decision Support) [Poess et al. 2002, Nambiar and Poess 2006], totaling 33 experimental runs on the same geo-distributed Kubernetes cluster.

This work makes three main contributions: *(i) Shuffle invariance under AQE* - across all 33 runs, shuffle read bytes vary by $<2\%$ across executor sizes at the same scale factor, because AQE's partition coalescing is data-driven, not executor-count-driven — refuting the primary empirical justification for vertical scaling in Spark 3.5 TPC-DS workloads; *(ii) Iso-resource parallel efficiency* - Small executors outperform Medium and Large by 22–23% at SF250 and SF500, with within-configuration efficiency of 62–90% vs. 36–56% relative to each configuration's own W2 baseline; *(iii) GC as the mechanism* - total GC time grows monotonically with executor heap size across every (SF$\times$W) point, reaching 84% higher for Large vs. Small at SF250/W6 — the dominant cost of vertical scaling.

The remainder of this paper is organized as follows. Section 2 reviews TPC-DS, Apache Spark, and AQE. Section 3 describes the infrastructure and experimental design. Section 4 contextualizes related work. Section 5 presents the three main findings (shuffle behavior, parallel efficiency, and GC overhead). Section 6 discusses threats to validity, and Section 7 concludes.

## 2. Background

This section presents the conceptual and technical foundations that support the analyses conducted in this work: the TPC-DS benchmark, the Apache Spark execution model with Adaptive Query Execution (AQE), and the executor configurations and efficiency metrics that underpin the iso-resource comparisons used throughout the paper.

### 2.1. TPC-DS Decision Support Benchmark

TPC-DS [Transaction Processing Performance Council 2021, Nambiar and Poess 2006] is the industry-standard OLAP benchmark, modeling decision-support workloads with 99 SQL queries over a snowflake schema (fact tables `store_sales`, `web_sales`, `catalog_sales`, plus dimensions). The scale factor (SF) parameterizes data volume from sub-gigabyte to multi-terabyte. The queries combine multi-way joins, `GROUP BY` aggregations, correlated subqueries, and window functions, making the benchmark shuffle-intensive and a rigorous stress test for resource management.

### 2.2. Apache Spark and Adaptive Query Execution

Apache Spark [Zaharia et al. 2016] executes query plans as directed acyclic graphs over distributed datasets. The *shuffle* stage redistributes data across executors by key, dominating network I/O in analytical workloads. Spark 3.0 introduced *Adaptive Query Execution* (AQE) [Zhu et al. 2020], which adapts plans at runtime using statistics from completed shuffle stages: (1) *partition coalescing* merges small post-shuffle partitions; (2) *broadcast*

*join conversion* swaps sort-merge for broadcast joins when one side is small; (3) *skew join handling* splits oversized partitions. Crucially, all three decisions are driven by runtime data statistics, not by executor count or memory size — with the implications analyzed in Section 5.1.

### 2.3. Executor Sizing and Parallel Efficiency

We define three executor configurations — Small (4c/14 GB), Medium (8c/28 GB), Large (12c/48 GB) — corresponding to $1\times$, $2\times$, $3\times$ the small unit, enabling iso-resource comparisons (e.g., Small$\times$8 = Medium$\times$4 = 32 cores, 112 GB). Parallel efficiency uses each configuration's own W2 baseline, as shown in Equation 1:

$$E(n) = \frac{T_{n_0}}{T_n} \cdot \frac{n_0}{n} \tag{1}$$

where $E = 1$ is linear scaling and $E < 1$ reflects coordination overhead, GC pauses, or serialization barriers. Since W2 baselines differ across configurations (Small/W2 = 8 cores vs. Medium/W2 = 16), cross configuration comparisons use the iso-resource framing of Section 5.2.

## 3. Infrastructure and Methodology

All experiments were executed on a Kubernetes v1.27.7 cluster of the Brazilian National Research and Education Network (RNP/LaMEX), with nodes spanning multiple Brazilian states. Table 1 summarizes the node groups available during the experiments.

**Table 1. Node groups in the geo-distributed Kubernetes cluster (RNP).**

| Group | Count | CPU/node | RAM/node | Status |
|---|---|---|---|---|
| Large (ids-es, ids-go, ids-rn) | 3 | 48 | 187 GiB | Ready |
| Large (ids-pb) | 1 | 24 | 172 GiB | Ready |
| Small (vm1-*, whx-*) | 6 | 8 | 15–16 GiB | Ready |
| **Total available** | **10** | - | **≈843 GiB** | - |

Figure 1 illustrates the benchmark architecture. Object storage is provided by a MinIO [MinIO, Inc. 2024] instance deployed within the cluster (S3-compatible API). TPC-DS data is stored in Delta Lake [Armbrust et al. 2020] format (Parquet files). Spark operates in `k8s://` mode without a SparkOperator, using a namespace-scoped service account (`ns-admin`). MinIO throughput was monitored throughout and remained in the 50–96 MB/s range, well below saturation, ruling out storage I/O as a confounding bottleneck.

**Pod scheduling policy.** All pods used the default `kube-scheduler` with a single soft `preferredDuringSchedulingIgnoredDuringExecution` affinity for the four large `ids-*` nodes; no hard `nodeSelector`, taints/tolerations, or `podAntiAffinity` were configured, and the affinity rule is identical for Small, Medium, and Large executors. The preference is a practical constraint: the smaller `vm1-*`/`whx-*` nodes (15–16 GiB RAM) cannot host even a Small executor's 14 GB

heap. All three configurations therefore draw from the same four-node pool, so a per-configuration performance gap cannot be explained by Small being scheduled on better hardware.

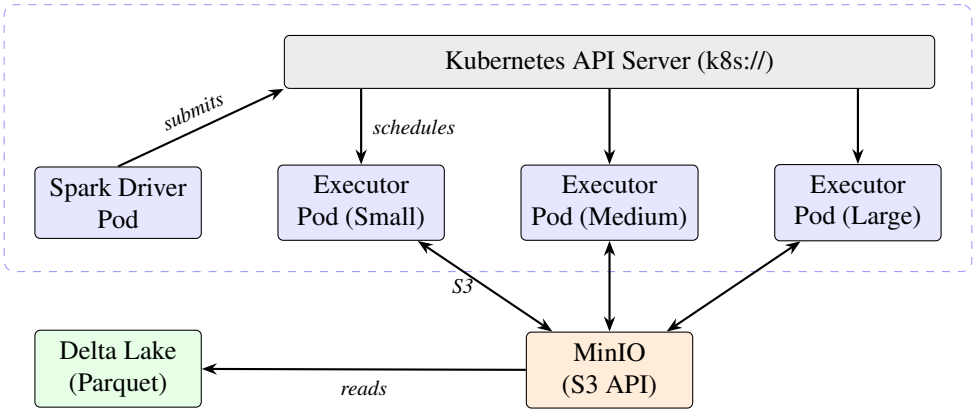


**Figure 1. Benchmark architecture on the RNP/LaMEX cluster. The Spark driver submits to the Kubernetes API (`k8s://`), which schedules executor pods with one of three sizing configurations (Small/Medium/Large). All executor pods read TPC-DS data from MinIO via the S3 API; data is stored in Delta Lake format.**

### 3.1. Experimental Design

Table 2 summarizes the experiment parameters. The three executor configurations are designed to enable *iso-resource comparisons*: configurations sharing the same total core and memory budget (e.g., Small×8 = Medium×4 = 32 cores, 112 GB) differ only in how resources are distributed across executor processes.

**Table 2. Experiment parameters.**

| Parameter | Value |
|---|---|
| Scale factors | SF100, SF250, SF500 |
| Executor configurations | Small (4c/14 GB), Medium (8c/28 GB), Large (12c/48 GB) |
| Worker range | W2–W10 (config-dependent) |
| Per-query timeout | 1 800 s (30 min) |
| Repetitions | 1 per configuration |
| Total runs | 33 |
| Lakehouse format | Delta Lake 3.0.0 (Parquet) |
| Driver memory | 8 GB |
| Spark version | 3.5.0 (AQE enabled by default) |
| Delta Lake version | 3.0.0 |
| MinIO version | RELEASE.2024-01-18 |

Table 3 shows the full experiment matrix. SF500/Small/W2 was excluded from the design by resource-management principle: SF250/Small/W2 already required over 5 h to complete, and projections for SF500/Small/W2 exceeded 10 h, which would have consumed the shared-cluster time budget without contributing informative data to the efficiency analysis (SF500/Small uses W4 as its baseline throughout Section 5.2). Large configurations start at W2 due to the higher per-executor resource cost.

**Table 3. Experiment matrix. Checkmarks indicate executed runs.**

| Config | W2 | W4 | W6 | W8 | W10 |
|---|---|---|---|---|---|
| SF100 / Small | ✓ | ✓ | ✓ | ✓ | - |
| SF100 / Medium | ✓ | ✓ | ✓ | ✓ | - |
| SF100 / Large | ✓ | ✓ | ✓ | - | - |
| SF250 / Small | ✓ | ✓ | ✓ | ✓ | - |
| SF250 / Medium | ✓ | ✓ | ✓ | ✓ | - |
| SF250 / Large | ✓ | ✓ | ✓ | - | - |
| SF500 / Small | - | ✓ | ✓ | ✓ | ✓ |
| SF500 / Medium | ✓ | ✓ | ✓ | ✓ | - |
| SF500 / Large | ✓ | ✓ | ✓ | - | - |

### 3.2. Metrics Collection

For each experimental run, we collected a comprehensive set of metrics to capture both query-level and system-level behavior. Specifically, we recorded: (i) per-query execution time (in milliseconds) along with execution status (*success* or *timeout*); (ii) stage-level metrics from Apache Spark, retrieved via the REST API (`localhost:4040`), including shuffle read and write volumes, executor run time, and garbage collection (GC) time per stage; and (iii) storage throughput metrics from MinIO, obtained through its Prometheus endpoint (`/minio/v2/metrics/cluster`). For the latter, snapshots were collected before and after each benchmark execution to compute net bytes read and total request counts.

All metrics were automatically persisted as structured JSON files in MinIO object storage by the Spark driver, ensuring traceability, reproducibility, and enabling independent post-processing and analysis.

**Timeout and failure handling.** A 1 800 s wall-clock timeout was enforced per query; on expiry the Spark job was cancelled, the query logged with `status="timeout"`, and execution proceeded to the next query. Aggregate times in Section 5 include the 30 min ceiling for every timed-out query in every configuration, so iso-resource comparisons sum identical workloads. In practice `q95` timed out in every (SF250, SF500) cell regardless of executor size, plus `q23` at SF250/Small/W2. No task-level failures (executor crashes, OOM, scheduler errors) occurred. The only excluded cell, SF500/Small/W2, was skipped *a priori* on resource-budget grounds (Table 3).

## 4. Related Work

Zhu et al. (2020) introduced AQE in Spark 3.0, demonstrating that runtime statistics enable more efficient query plans than static optimisation. That paper established AQE's correctness and average-case speedup, but did not examine how AQE changes the economics of executor sizing. Our work fills this gap: by measuring shuffle bytes, parallel efficiency, and GC time across 33 configurations on real geo-distributed infrastructure, we provide empirical evidence that AQE's partition coalescing is data-driven and therefore eliminates the shuffle reduction argument for vertical scaling that pre-AQE analyses assumed [Tang et al. 2016, Shi et al. 2015].

Armbrust et al. (2021) establish the lakehouse architecture and identify executor-level resource management as an open problem; Behm et al. (2022) improve intra-

executor CPU efficiency via vectorisation (Photon) but do not address executor sizing; Huang et al. (2024) compare lakehouse formats with TPC-DS in a single-machine Docker environment without scaling analysis.

Li et al. (2022) compare Spark on Kubernetes vs. YARN; Abe et al. (2023) evaluate Spark on Kubernetes under varied resource configurations; Guo et al. (2023) study cloud-native autoscaling. None isolates executor heap size as an independent variable, measures per-stage GC overhead at scale, or quantifies the AQE-vs-packaging interaction — which is the contribution of the present work.

The Spark documentation recommends avoiding executors with more than 5 cores to limit HDFS I/O contention [Zaharia et al. 2016]. This heuristic predates AQE and does not account for modern shuffle partition coalescing. Our findings empirically validate that the "prefer smaller executors" heuristic holds in the Spark 3.5 era, but for a different reason: not HDFS I/O contention, but JVM GC overhead under sustained analytical load.

Tang et al. (2016) characterize TPC-DS query execution bottlenecks on Spark, identifying shuffle as the dominant cost for complex multi-join queries. Shi et al. (2015) compare Spark and MapReduce on TPC-DS, finding that Spark's in-memory model outperforms MapReduce for iterative and aggregation-heavy queries.

Neither study examines executor sizing or scaling strategies. Our work adds a systematic comparison of executor sizing across three scale factors with per-stage GC and shuffle metrics collected via the Spark REST API on real geo-distributed infrastructure.

## 5. Results

This section presents the experimental results from the 99 TPC-DS queries across the executor configurations and scale factors, organized around three findings: shuffle volume is essentially invariant to executor size under AQE; horizontal scaling with smaller executors achieves higher parallel efficiency under iso-resource conditions; and JVM GC overhead grows with executor heap size, becoming the dominant cost of vertical scaling.

### 5.1. Finding 1: Shuffle Volume Is Invariant to Executor Size

The prevailing argument for vertical scaling in Spark is that larger executors process more data per task, leading to fewer shuffle partitions and therefore less data redistributed across the network. Our results directly contradict this assumption.

Figure 2 shows total shuffle read bytes aggregated across all 99 TPC-DS queries for each configuration. At SF100, all three executor types produce $\approx$94 GB of shuffle reads, regardless of whether resources are allocated as 8 small executors, 4 medium, or 3 large. The pattern holds at SF250 ($\approx$268 GB) and SF500 ($\approx$327 GB): shuffle volume varies by less than 2% across configurations at the same scale factor, and in no single (SF, worker-count) combination does any configuration deviate by more than 2% from the mean shuffle volume at that scale factor.

AQE's partition coalescing reduces the number of post-shuffle tasks but does not reduce the bytes written: total shuffle volume is determined by the logical plan (join keys, aggregation predicates) and dataset cardinality, both fixed by the scale factor. Broadcast-join conversion and skew handling further make AQE's behavior data-driven rather than executor-count-driven [Zhu et al. 2020]. The static intuition "fewer executors $\Rightarrow$ fewer

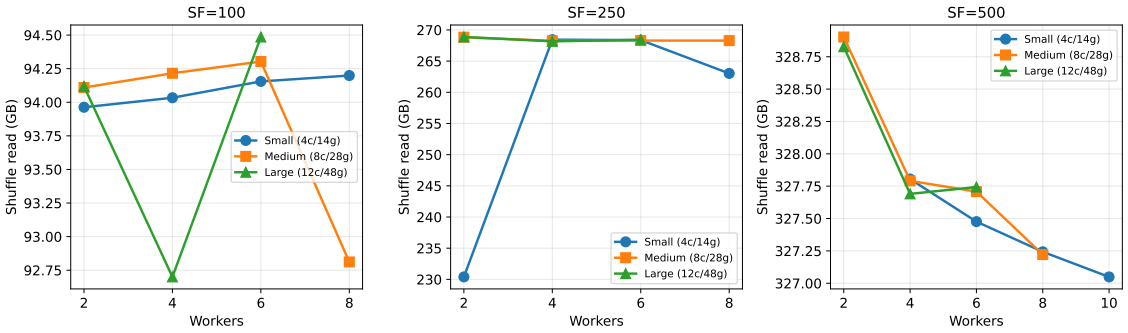


**Figure 2. Total shuffle read (GB) aggregated across all 99 TPC-DS queries, by executor configuration and worker count. Shuffle volume varies by <2% across Small, Medium, and Large executors at each scale factor across all 33 runs, confirming executor-size invariance.**

partitions ⇒ less shuffle volume" therefore does not hold under AQE, and any vertical-scaling advantage must come from factors other than shuffle reduction — examined in the following sections.

### 5.2. Finding 2: Horizontal Scaling Achieves Higher Efficiency

Given that shuffle volume is invariant across executor sizes, we compare execution time and parallel efficiency across configurations.

#### 5.2.1. Iso-Resource Comparison

The cleanest way to compare executor sizes is to hold total resources constant and vary only how they are packaged. Figure 3 shows execution time for configurations with approximately 32–48 total cores. At SF250, Small×8 (32 cores, 112 GB) completes in 162 min, while Medium×4 (32 cores, 112 GB) takes 199 min and Large×4 (48 cores, 192 GB) takes 198 min - a 23% and 22% penalty respectively for the same or more resources. At SF500, the pattern persists: Small×8 (250 min) outperforms Medium×4 (274 min) and Large×4 (273 min) by approximately 9%. The narrowing relative gap at SF500 compared with SF250 (9% vs. 22–23%) is consistent with both executor types facing heavier absolute GC pressure at higher data volume: the absolute time advantage of Small executors remains substantial (24 min at the SF500 iso-resource point), but as total runtime grows, the GC-induced penalty represents a smaller fraction of the larger denominator.

**Statistical strength.** Treating the 99 TPC-DS queries as paired observations within each cell: at SF250 Small beats Medium on 85/98 queries (median speedup 22.7%) and Large on 84/98 (median 24.3%), Wilcoxon $p < 5 \times 10^{-14}$ in both cases; a 10 000-resample bootstrap places the total-time gap at 22.0% [13.4%, 29.2%] vs. Medium and 21.5% [10.2%, 30.1%] vs. Large (95% CIs). At SF500 the effect is smaller but consistent: Small wins on 74–76/98 queries, gap 9.9% [6.8%, 13.6%] and 9.7% [5.2%, 14.4%], Wilcoxon $p < 2 \times 10^{-7}$. Cross-run variability is addressed in Section 6.

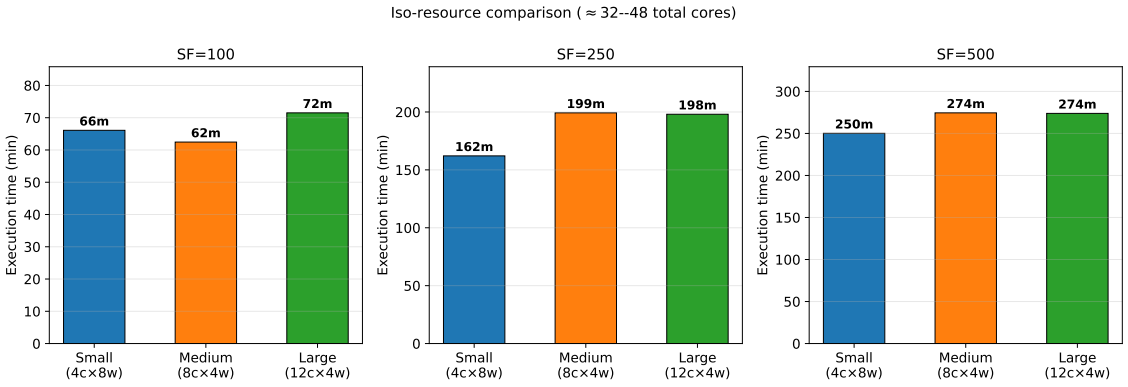


**Figure 3. Execution time for iso-resource configurations (≈32–48 total cores). Small executors consistently outperform Medium and Large at equal or smaller resource budgets. The Large×4 configuration uses more total cores than Small×8 yet takes longer.**

### 5.2.2. Parallel Efficiency

Figure 4 and Table 4 show parallel efficiency as workers are added within each configuration, using W2 as baseline. Note that these efficiency values measure how well each configuration uses *additional executors of its own type* - the baselines are not iso-resource (Small/W2 = 8 cores; Medium/W2 = 16 cores; Large/W2 = 24 cores). Cross-configuration comparisons are made via the iso-resource analysis above.

Within each configuration, the efficiency gap is pronounced. At SF250, Small reaches W4 with 90% efficiency; Medium drops to 56% at the same doubling of its own worker count. By W8, Small retains 62% efficiency while Medium falls to 36%. Large configurations degrade even faster, reaching only 41% at W6 for SF250.

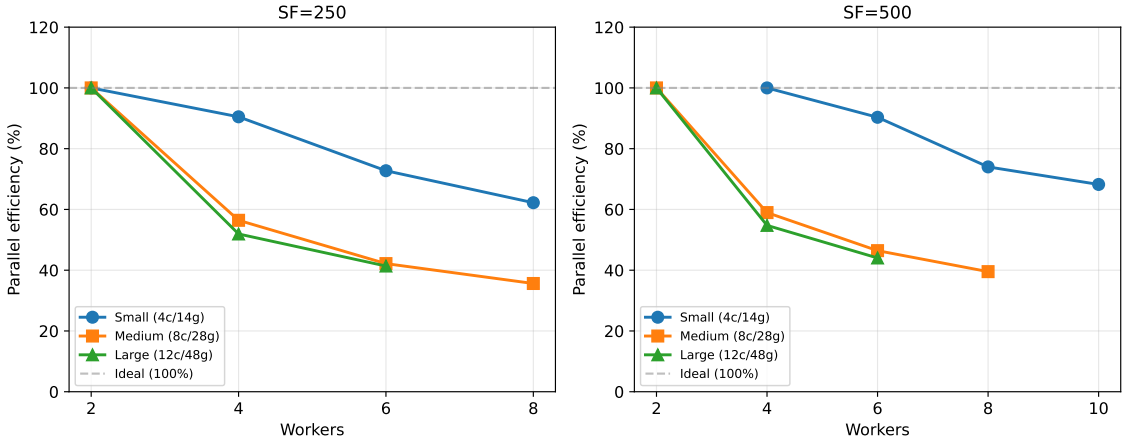


**Figure 4. Parallel efficiency (%) as worker count increases, relative to each configuration's own W2 baseline. Efficiency measures within-configuration scaling; cross-configuration comparisons require the iso-resource framing of Figure 3.**

The full execution time curves are shown in Figure 5. Small configurations benefit more from each additional worker because each executor manages a smaller JVM heap

(reducing GC pressure, analyzed in Section 5.3), incurs less intra-JVM scheduling contention per task slot, and exposes more independent task parallelism to the Spark driver.

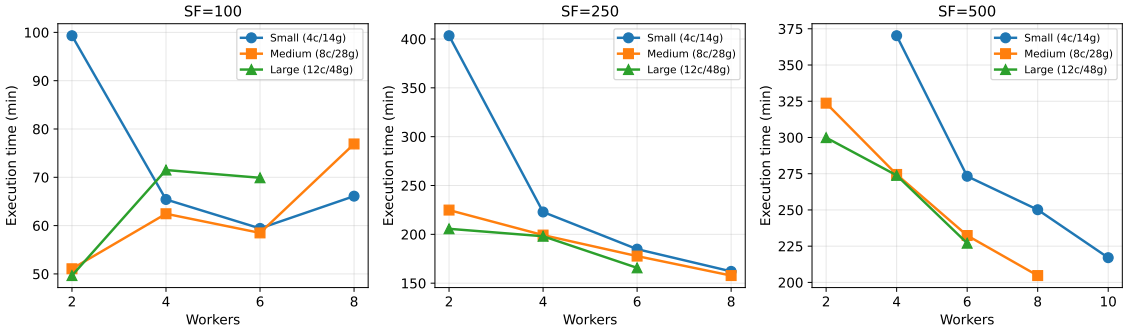


**Figure 5. Total TPC-DS execution time (minutes) by worker count, scale factor, and executor configuration.**

**Table 4. Parallel efficiency at SF250 and SF500 by executor configuration (W2 baseline, Equation 1). Values reflect within-configuration scaling; see Section 5.2 for cross-configuration comparison.**

| | SF250 | | | SF500 | | |
|---|---|---|---|---|---|---|
| **Workers** | **Small** | **Medium** | **Large** | **Small** | **Medium** | **Large** |
| W2 (base) | 100% | 100% | 100% | - | 100% | 100% |
| W4 | 90% | 56% | 52% | - | 59% | 55% |
| W6 | 73% | 42% | 41% | 62% | 46% | 44% |
| W8 | 62% | 36% | - | 51% | 39% | - |
| W10 | - | - | - | 43% | - | - |

## 5.3. Finding 3: GC Overhead Grows With Executor Heap Size

To understand *why* vertical scaling underperforms even at iso-resource configurations, we examine garbage collection (GC) time reported by the Spark REST API per stage. Figure 6 shows total GC time accumulated across all 99 queries, summed across all executor threads for the full benchmark run.


**Figure 6. Cumulative GC time (hours) across all queries and all executor threads, by executor configuration and worker count. GC time grows monotonically with executor heap size across all scale factors and worker counts.**

At SF250/W6, Small executors accumulate 4.9 h of total GC time; Medium accumulates 6.5 h; Large accumulates 9.0 h - an 84% increase from Small to Large despite

executing the *same queries on the same data*. At SF500/W6, the values are 8.9 h (Small), 10.7 h (Medium), and 13.6 h (Large). The growth is monotonic: no configuration at any scale factor inverts the ordering.

Spark 3.5 uses G1GC by default. Two mechanisms drive larger-heap cost: a *structural* effect, where mixed-GC cycles must evacuate more live regions as heap size grows independent of allocation rate; and a *workload-dependent* effect, where sustained shuffle materialization in large executors can push the allocation rate past G1's concurrent marking throughput, triggering both more frequent and longer cycles. GC pauses are particularly damaging because they suspend executor threads entirely, stalling tasks and risking driver heartbeat timeouts; a 48 GB heap concentrates intermediate shuffle state in a single JVM, amplifying both effects.

**Could inter-regional latency drive the penalty?** Two facts argue against attributing the configuration ordering to geo-distributed shuffle rather than GC. First, latency acts on shuffle *volume*, which Section 5.1 showed is invariant across executor sizes ($<2\%$); a latency-driven slowdown would scale all configurations proportionally, not produce the monotonic Small $<$ Medium $<$ Large ordering reported here. Second, the GC-time delta alone covers most of the execution-time gap: at SF250/W6, Large accumulates 4.1 h more GC time than Small. The absolute magnitudes are still influenced by RNP's network topology; Section 6 addresses external validity.

**Query heterogeneity.** Not all TPC-DS queries are equally affected by executor sizing. Simple scan-and-aggregate queries (e.g., q1, q7, q19) complete quickly under all configurations and show small absolute time differences between Small and Large. The GC penalty is concentrated in long-running, shuffle-intensive queries such as q23a/b (nested CTEs over `store_sales`) and q14a/b (correlated subqueries), which at SF250 take 12–22 minutes per run and account for the bulk of the GC accumulation shown in Figure 6. The 84% GC overhead reported is thus a workload-level summary dominated by the heaviest queries; for an all-simple-query workload, the vertical scaling penalty would be smaller.

**Implication:** In Spark on JVM, concentrating resources into fewer large executors trades distributed execution overhead for JVM GC overhead. For workloads with sustained shuffle materialization (such as TPC-DS), the GC penalty dominates any benefits from reduced inter-executor coordination, and this effect grows with heap size.

## 6. Threats to Validity

This section discusses threats to the validity of the experimental results and the measures adopted to mitigate them, covering both internal validity (reliability of the experimental procedure and measurements) and external validity (generalizability to other infrastructures, workloads, and engines).

### 6.1. Internal Validity

Each cell of the matrix was executed once (72 node-hours total). Three lines of evidence support the directional findings. *(i) Paired tests.* The 22% iso-resource gap rests on 99 per-query measurements per cell: Wilcoxon signed-rank gives $p < 5 \times 10^{-14}$ at SF250

and $p < 2 \times 10^{-7}$ at SF500 (Small faster on 74–85/98 queries); non-parametric bootstrap 95% CIs on the total-time gap exclude zero at every point — SF250: [13.4%, 29.2%] vs. Medium and [10.2%, 30.1%] vs. Large; SF500: [6.8%, 13.6%] and [5.2%, 14.4%] respectively. *(ii) Reproducibility.* Two SF100 cells were independently re-executed: SF100/Small/W2 CV = 9.2%, SF100/Medium/W4 CV = 5.4%. The SF250 gaps (21–24%) are well outside this noise floor; the SF500 gap ($\approx$10%) is closer to it and is disclosed as such. *(iii) Structural consistency.* Shuffle invariance ($<$2%) holds across all 33 runs, and GC monotonicity (Small $<$ Medium $<$ Large) holds at every (SF$\times$W) comparison point. Independent repetition of every cell remains desirable, particularly at SF500.

All executor sizes share the same four-node pool under identical scheduling (Section 3), so the differential hardware-bias hypothesis cannot explain the configuration ordering; within-pool node heterogeneity contributes to run-to-run noise as quantified by the SF100 CVs above. MinIO throughput (50–96 MB/s) remained below saturation.

### 6.2. External Validity

Results reflect the specific topology and inter-node latency of the RNP geo-distributed network. On homogeneous, low-latency public-cloud clusters (e.g., AWS EKS, GCP GKE), the magnitude of efficiency differences may vary, though the shuffle invariance result is a property of AQE's design and should generalize across cluster topologies. Findings are specific to Spark 3.5.0 with AQE enabled and Delta Lake; other execution engines (e.g., Trino, Flink) or earlier Spark versions (pre-AQE) may exhibit different shuffle behavior. Generalization beyond TPC-DS (e.g., TPC-H, streaming workloads) requires additional evaluation.

## 7. Conclusion

Vertical executor scaling offers no advantage for TPC-DS workloads on Spark 3.5 with AQE: data-driven partition coalescing eliminates the shuffle-reduction argument for large executors, and concentrating memory into fewer JVMs only amplifies G1GC pressure (up to 84% more GC time), while horizontal scaling achieves 22% shorter execution at equal resource budgets. The practical recommendation is therefore to allocate resources as small executors ($\sim$4 cores, 14 GB) and scale out. This applies to JVM-based Spark with sustained shuffle materialization; workloads with minimal shuffle or memory-resident datasets may exhibit a different trade-off.

Future work spans four directions: (i) evaluating alternative GC strategies (ZGC, Shenandoah) to assess whether tuning can mitigate vertical-scaling losses; (ii) characterising the per-query-class sensitivity to executor sizing; (iii) replicating the analysis on public-cloud Kubernetes (e.g., AWS EKS) to isolate network-topology effects; and (iv) feeding the empirical evidence into a workload-profile-driven executor-sizing recommender, automating the horizontal-vs-vertical choice.

## ACKNOWLEDGEMENTS

The results reported in this work were significantly influenced by experiments carried out at RNP's National Multi-User Laboratory for ICT Experimentation [Pedrosa et al. 2024, Mondin and Dias 2023], and supported by CNPq, Brazil (Productivity in Research fellowship – Grant No. 310043/2025-5).

## References

Abe, H. et al. (2023). Evaluating Spark on Kubernetes under different resource configurations. In *2023 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE.

Armbrust, M., Das, T., Torres, L., Yavuz, B., Zhu, S., Xuan, R., Ghodsi, A., Stoica, I., and Zaharia, M. (2020). Delta Lake: High-performance ACID table storage over cloud object stores. *Proceedings of the VLDB Endowment*, 13(12):3411–3424.

Armbrust, M., Ghodsi, A., Xin, R., and Zaharia, M. (2021). Lakehouse: A new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*.

Behm, A., Palkar, S., Agarwal, U., Armbrust, M., Cashman, M., Chakraborty, A., Chenghao, A., Das, T., Davidson, A., Dhar, A., Ghodsi, A., Gupta, S., Jaeger, R., Khalapyan, L., Lai, E., Liu, S., Luszczak, A., Sankaranarayanan, H., Soman, B., Stuhrm, T., Sundaram, S., Tsai, A., Wang, Y., Wang, Y., Wu, R., Yavuz, B., Stoica, I., Boncz, P. A., and Zaharia, M. (2022). Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 ACM SIGMOD International Conference on Management of Data*, pages 2326–2339.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, Omega, and Kubernetes. *ACM Queue*, 14(1):70–93.

Guo, W. et al. (2023). Optimizing analytical workloads on cloud-native Spark. In *2023 IEEE International Conference on Cloud Computing*. IEEE.

Huang, G., Chen, Z., et al. (2024). Benchmarking the data lake ecosystem: Delta Lake, Apache Iceberg, and Apache Hudi. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Li, W. et al. (2022). Performance evaluation of Apache Spark on Kubernetes. In *2022 IEEE International Conference on Big Data*, pages 1–8. IEEE.

MinIO, Inc. (2024). MinIO: High performance object storage.

Mondin, L. and Dias, G. (2023). Plataformas flexíveis para experimentação: o caminho para atender as novas demandas de experimentação científica em TICs. In *Anais do XIV Workshop de Pesquisa Experimental da Internet do Futuro (WPEIF)*, pages 25–32, Brasília/DF. Sociedade Brasileira de Computação.

Nambiar, R. O. and Poess, M. (2006). The making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1049–1058.

Pedrosa, E., Martins, J., Sousa, F., Mondin, L., and Dias, G. (2024). Indo além das simulações com o serviço de testbeds: Ambientes reais para experimentação científica em TICs. In *Anais do XV Workshop de Pesquisa Experimental da Internet do Futuro (WPEIF)*, pages 47–54, Niterói/RJ. Sociedade Brasileira de Computação.

Poess, M., Smith, B., Kollar, L., and Larson, P. (2002). Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, page 582–587, New York, NY, USA. Association for Computing Machinery.

Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Ozcan, F. (2015). Clash of the titans: MapReduce vs. Spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121.

Tang, Z. et al. (2016). Benchmarking Apache Spark and Hadoop MapReduce on big data workloads. In *2016 IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE.

Transaction Processing Performance Council (2021). TPC-DS decision support benchmark, version 3.2.0. Technical report, TPC.

Zaharia, M., Xin, R. S., Wendell, P., Das, T., Armbrust, M., Dave, A., Meng, X., Rosen, J., Venkataraman, S., Franklin, M. J., et al. (2016). Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59(11):56–65.

Zhu, C., Han, W., Huang, L., and Ma, Y. (2020). Adaptive query execution: Speeding up Spark SQL at runtime. *Proceedings of the VLDB Endowment*, 13(12):3533–3546.