JOURNAL OF
Emerging Paradigms
in Computing Systems
JEPCS

Journal of Emerging Paradigms in Computing Systems (JEPCS) | Volume 01, Issue No. 01, 2025

ARTICLE TYPE: Research Article

Article ID: JEPCS-2025-0101-0010

Robustness of PostgreSQL Query Optimization Under Stale Statistics, Data Drift, and Distribution Skew: A Depth-and-Breadth Study

Sadia Aslam[1,*] and Muhammad Umar Hameed[1]

[1] Department of Computer Science, Faculty of Information Technology & Computer Science, University of Central Punjab, Lahore, Pakistan.

* Corresponding author: sadia.aslam@ucp.edu.pk

Received: September 17, 2024 **Accepted:** December 07, 2024

Abstract- PostgreSQL relies on planner statistics to estimate cardinalities and compare candidate execution plans. When updates change value distributions, these statistics can continue to describe the pre-update data. We examine the runtime effect of this mismatch in PostgreSQL 16.4 using paired database copies with identical current data. The stale copy retains statistics collected before controlled changes to non-key values, whereas the fresh copy runs `ANALYZE` after the same changes. The depth phase evaluates five representative TPC-H scale-factor-1 query forms and eight Join Order Benchmark (JOB) queries at 20%, 50%, and 80% targeted drift. For each depth query-condition, we use two warm-up runs followed by seven measured executions. The breadth phase evaluates all 113 canonical JOB queries at 80% drift using one warm-up run and three measured executions. Across 106 uncensored stale/fresh pairs, the median runtime ratio is 1.006 (bootstrap 95% CI: 0.992-1.044), while the one-sided Wilcoxon signed-rank test gives $p = 0.027$. Thirty-one of 113 queries are at least 2× slower, 14 are at least 10× slower, and three are at least 100× slower. Seven stale conditions exceed the 180 s limit, whereas no fresh condition times out. JOB 2c has the largest exact ratio at 3660×. In selected high-ratio cases, the retained plans show large cardinality underestimates together with changes in join topology. These results show that the largest stale-statistics slowdowns occur in a subset of the workload even though the workload median remains close to parity.

Keywords- PostgreSQL; query optimization; cardinality estimation; stale statistics; data drift

1. Introduction

Before an SQL statement is executed, the query optimizer selects a physical execution plan from alternative access paths, join orders, and physical operators. Cost-based optimizers compare these alternatives using estimated execution costs, an approach established by the System R optimizer [1]. Cardinality estimates for intermediate results are an important input to these cost calculations [2]. When an intermediate cardinality is estimated incorrectly, the error can propagate to later joins that depend on that result [3]. The resulting estimates can change the relative costs assigned to candidate plans and, consequently, the plan selected for execution. Such errors can change the relative costs assigned to candidate plans and, consequently, the plan selected for execution.

PostgreSQL stores planner statistics in its system catalog and uses them when estimating query cardinalities. The `ANALYZE` command samples table contents and refreshes these statistics for subsequent query planning [4]. The `ANALYZE` command samples table contents and updates the planner statistics used during query optimization [4]. These statistics describe the data at the time they are collected. They can therefore become inaccurate after the underlying values change. A bulk update, data migration, ingestion batch, or a concentrated change in a categorical value may alter the current distribution while the planner continues to use statistics collected from the earlier state. PostgreSQL documentation

identifies inaccurate planner statistics as one possible cause of poor plan selection and provides automatic and manual analysis for refreshing them [5].

Research on query optimization has examined cardinality estimation, join ordering, cost models, and the runtime effect of different optimizer components. The Join Order Benchmark (JOB) was introduced with multi-join queries over IMDb data containing correlations that are difficult for conventional optimizers [6]. A later JOB study further examined the effect of correlations that cross join boundaries [7]. These studies provide evidence that estimation errors can affect the plans selected by an optimizer. The setting considered in this paper is more specific. We keep PostgreSQL and the SQL workload unchanged, modify the data distribution, and then examine what happens when the planner statistics still describe the data before that modification.

The effect of such a mismatch can depend on the query. If the stale estimates do not change the selected access path or join strategy, the execution may remain close to the result obtained with refreshed statistics. For another query, the same type of distribution change can alter the estimated cost of competing plans and lead to a different join order or physical operator. We therefore examine query-level runtimes rather than relying only on a single workload average or median. This is especially useful for identifying a small number of queries whose behavior differs substantially from the rest of the workload.

In this paper, we study stale planner statistics by creating paired PostgreSQL database copies from the same analyzed database. Both copies receive the same deterministic modifications to non-key attributes and contain the same current data when the queries are executed. The difference is that the stale copy keeps the statistics collected before the update, whereas the fresh copy runs `ANALYZE` after receiving the same update. Primary and foreign keys are not modified, so the join relationships remain unchanged. We record execution time, node-level cardinality error, join and scan operators, planning time, coarse plan topology, and the duration of `ANALYZE`. Buffer counters are also retained in the experimental output. Their node-wise sums are not used as quantitative I/O evidence because the captured PostgreSQL counters can be cumulative across plan nodes.

The evaluation is carried out in two phases. The first is a depth experiment with five representative TPC-H SF1 query forms and eight JOB queries. These queries are executed at 20%, 50%, and 80% targeted distribution drift, using two warm-up executions and seven measured executions for each successful condition. This phase is used to follow the same queries as the amount of drift changes and to inspect their PostgreSQL plans in detail. The second phase evaluates all 113 canonical JOB queries at 80% drift, using one warm-up and three measured executions. Before inspecting the complete 113-query outcome, we fixed the query population, repetition count, timeout treatment, slowdown thresholds, and statistical tests for this phase. The protocol was fixed internally for the experiment and was not externally preregistered.

The breadth experiment shows that the effect is not uniform across JOB. For the 106 uncensored stale/fresh query pairs, the median runtime ratio is 1.006×, with a bootstrap 95% confidence interval of 0.992-1.044. At the query level, 31 of 113 queries are at least 2× slower with stale statistics. Fourteen are at least 10× slower, and three are at least 100× slower. Seven stale query conditions exceed the 180 s statement cap, whereas every corresponding fresh condition completes within the cap. JOB 2c gives the largest exact ratio at 3660×. The depth experiment also shows increasing sensitivity in the same JOB family as the targeted drift is increased from 20% to 80%.

The work provides an empirical characterization of this behavior in PostgreSQL rather than a new optimizer or cardinality estimator. The paired databases allow statistics freshness to change while the current data and SQL remain fixed. Repeated measurements in the depth phase are used to examine how individual plans change with drift, while the complete JOB breadth phase measures the frequency of large slowdowns at the severe drift setting. We also retain the PostgreSQL JSON plans for the largest regressions. These plans are used to compare estimated and actual rows and to examine the join operators selected under stale and fresh statistics.

2. Background and Related Work

2.1. Cost-Based Optimization and Cardinality Error

Cardinality estimates are used throughout cost-based query optimization. The estimated output of one operator affects the cost assigned to operators that use that output later in the plan. An error at an early join can therefore propagate through the remaining join tree [3]. Its effect on execution time is not determined only by the numerical size of the error. If an estimate changes the relative cost ordering of candidate plans, the optimizer may select a different join order or join algorithm. The resulting runtime can then change because a different physical plan is executed.

Several approaches have been proposed to reduce dependence on estimates made before query execution. LEO uses the cardinalities observed during completed executions as feedback for subsequent optimization [8]. Progressive optimization follows a different mechanism. It compares estimated and observed cardinalities while a query is executing and can trigger re-optimization when the observations move outside the range assumed by the current plan [9]. PostgreSQL must still select its initial execution plan from the statistics available when planning starts. Our experiment examines what happens to this initial decision when those statistics no longer describe the current data distribution.

JOB is particularly relevant to cardinality estimation because its queries contain correlations across the IMDb schema. Leis et al. [6] reported large estimation errors for several JOB queries and examined how those errors affected optimizer plan quality. Their later analysis evaluated JOB under additional execution settings and again examined the interaction between cardinality estimation and query plans [7]. Other work has attempted to improve the estimates themselves. Index-based join sampling uses available indexes to obtain samples for estimating join cardinalities [10]. Kipf et al. [11] use learned models to estimate cardinalities for correlated joins. Yang et al. [12] model the joint distribution of table attributes using an autoregressive approach, while DeepDB constructs a learned probabilistic representation of the data [13]. In this paper, PostgreSQL's estimator is kept unchanged. We study the effect of allowing its stored planner statistics to become stale after the underlying data distribution changes.

2.2. Statistics, Updates, and Distribution Shift

PostgreSQL collects planner statistics through `ANALYZE` [4]. Standard column statistics summarize the observed data distribution, while extended statistics can represent selected dependencies among columns of the same table. PostgreSQL 16 documentation states that extended statistics are not currently used when estimating the selectivity of table joins [14]. Thus, this mechanism does not directly provide join-selectivity information for dependencies that occur across different tables. A separate problem occurs when the values in a table change after the statistics have been collected. In that case, even the stored single-table summaries may no longer represent the current value frequencies.

The estimator is kept the same in both conditions of our experiment. We first modify the data distribution in two database copies using the same deterministic updates. One copy retains the statistics collected before the update, while the other copy runs `ANALYZE` after the update. Both copies then execute the same SQL over the same current table values. This paired setup allows us to examine how refreshing PostgreSQL's own planner statistics affects cardinality estimates, plan selection, and runtime after the data distribution has changed.

Distribution shift has also been examined for learned database components. Zeighami and Shahabi analyze learned database operations, including cardinality estimation, when the deployment distribution differs from the distribution used to build the model [15]. Roq considers uncertainty in a learned cost model when selecting query plans [16]. These methods use different internal representations from PostgreSQL statistics, but they face a related deployment problem: information used by the optimizer may no longer match the current data distribution. Our experiments study this issue using PostgreSQL's built-in statistics rather than a learned estimator.

2.3. Runtime and Q-Error

We use Q-error to measure the multiplicative difference between estimated and actual row counts. To avoid division by zero, the implementation replaces each count by at least one:

$$Q(e, a) = \max\left(\frac{\max(e, 1)}{\max(a, 1)}, \frac{\max(a, 1)}{\max(e, 1)}\right), \tag{1}$$

Here, e denotes estimated rows and a denotes actual rows. $Q = 1$ represents an exact cardinality estimate. Values greater than 1 indicate the multiplicative difference between the estimated and actual row counts.

Q-error is useful for identifying estimation errors, but its value alone does not determine the quality of the final execution plan. An error at one plan node can affect plan selection differently from an error of the same magnitude at another node. Flow-Loss accounts for this distinction by considering how cardinality estimates influence plan search [17]. Han et al. [18] also evaluate the limitations of using Q-error alone as an indicator of end-to-end plan quality. For this reason, execution time is used as the main end-to-end measurement in our experiments, while Q-error is used to inspect the estimation behavior inside a plan. For every successful execution, we calculate the nearest-rank 90th percentile of node-level Q-error. We use this node-level summary rather than relying only on the root estimate because large estimation errors can occur at intermediate joins below the root node.

3. Research Questions and Experimental Design

The two experimental phases answer three research questions. RQ1 follows the selected queries as the amount of targeted drift increases, while RQ2 examines slowdown prevalence across the complete 113-query JOB workload. RQ3 uses the retained execution plans to examine the queries with the largest stale/fresh runtime regressions. The smaller depth query set is therefore used for repeated plan analysis and is not used to estimate slowdown prevalence across JOB.

RQ1: How does sensitivity to stale statistics change as targeted distribution drift increases from 20% to 80%?

RQ2: How prevalent are stale-statistics slowdowns across all 113 canonical JOB queries at the 80% drift setting used in the breadth phase?

RQ3: Which cardinality-estimation and plan changes accompany the largest stale/fresh runtime regressions?

Table 1: *Two-phase experimental design.*

Item	Depth phase	Breadth phase
Engine	PostgreSQL 16.4	PostgreSQL 16.4
Data	TPC-H SF1 + JOB/IMDb	JOB/IMDb
Query population	5 TPC-H forms + 8 JOB queries	All 113 canonical JOB queries
Targeted drift	20%, 50%, 80%	80%
Conditions	Baseline, stale, fresh	Baseline, stale80, fresh80
Warm-up / measured	2 / 7	1 / 3
Statement cap	180 s, right-censored	180 s, right-censored
Role	Mechanism and severity	Full-workload prevalence

Table 1 summarizes how the two phases are configured. The depth phase was completed first, and its results were used to select the 80% drift setting for the full JOB breadth experiment. Before examining the complete 113-query breadth outcome, we fixed the query population and repetition schedule together with the timeout treatment and slowdown thresholds. The Wilcoxon test, bootstrap interval, and family-level summaries were also specified at that stage. This protocol was internal to the study and was not registered externally.

For each stale/fresh comparison, the databases are created from the same analyzed base. The two copies receive identical deterministic updates. We then run `ANALYZE` only on the fresh copy. The paired queries therefore execute the same SQL on the same current table values, while the planner uses different statistical information in the two conditions.

4. Methodology

4.1. Platform and Datasets

We ran the experiments on an Ubuntu 24.04 GitHub Actions hosted runner using the `postgres:16.4` container image. PostgreSQL 16.4 was released on 8 August 2024 [19]. JIT was disabled for the measured queries, and `max_parallel_workers_per_gather` was set to 0. Autovacuum was disabled during the controlled experiment. The PostgreSQL server used `max_wal_size=2GB` and `checkpoint_timeout=30min`, while `shared_buffers` remained at the container default of 128 MB. After applying the distribution changes, we issued a forced `CHECKPOINT` before timing the queries. This separates the update phase from the timed query phase, although it does not remove all possible operating-system cache effects.

For JOB, we loaded the May-2013 IMDb snapshot associated with the public Join Order Benchmark. The database contains 21 tables. In the loaded database, `cast_info` has 36.24 million rows, `movie_info` has 14.84 million, `movie_keyword` has 4.52 million, and `title` has 2.53 million. We used the JOB schema and foreign-key index definitions released with the public benchmark. The workload consists of 113 canonical SQL queries grouped into 33 numbered families [6, 7].

The depth phase also includes TPC-H at scale factor 1 [20]. We generated the data with the TPC-H generator available through the DuckDB TPC-H extension and loaded the resulting tables into PostgreSQL. The loaded `lineitem` table contains 6,001,215 rows. The `orders`, `customer`, and `part` tables contain 1.5 million, 150,000, and 200,000 rows, respectively. We evaluate five query forms: q03, q06, q12, q14, and q19. These are hand-coded representative SQL forms rather than the complete official `qgen` workload. The PostgreSQL configuration also uses the index set defined for this experiment. We therefore use the TPC-H component only as a synthetic control workload and do not report it as an official TPC-H benchmark result.

4.2. Controlled Distribution Drift

The drift operation changes non-key attributes while leaving primary- and foreign-key values unchanged. For each modified JOB table, a row with identifier i is selected at drift level d when `id mod 100` < `d`. We set `company_name.country_code` to `[de]`, `keyword.keyword` to `character-name-in-title`, and `title.production_year` to 2010 for the selected rows. At $d = 80$, the rule selects identifiers whose modulo-100 remainder is between 0 and 79. This defines the 80% targeted-drift condition. The final fraction of rows containing an assigned value can be higher than the targeted fraction because some rows may already contain that value before the update.

The TPC-H intervention applies the same modulo threshold to relation-specific keys. For `customer`, rows selected through `c_custkey` have `c_mktsegment` set to `BUILDING`. In `lineitem`, the modulo condition on `l_orderkey` controls the rows whose `l_shipmode` is set to `MAIL`. A separate condition on `l_partkey` sets `l_discount` to 0.06. For `part`,

Table 2: *Depth-phase aggregate outcomes.*

Data	Drift	Median	p90	Median stale p90 Q	Median fresh p90 Q	Stale plan Δ	Fresh plan Δ
JOB	20%	1.011×	272.5×	225.69	34.49	0%	37.5%
JOB	50%	0.997×	653.5×	183.50	53.72	0%	37.5%
JOB	80%	1.045×	1195.7×	182.81	124.40	0%	50.0%
TPC-H	20%	0.931×	0.996×	3.50	2.83	0%	40.0%
TPC-H	50%	1.018×	1.249×	3.50	1.77	0%	60.0%
TPC-H	80%	1.021×	1.825×	3.50	1.21	20.0%	60.0%

rows selected through `p_partkey` have `p_brand` and `p_container` set to `Brand#12` and `SM BOX`. These rules are applied at 20%, 50%, and 80% targeted drift in the depth phase.

We run `ANALYZE` on the base database before creating the experimental copies. The stale and fresh copies are then cloned from this same analyzed state and receive identical deterministic updates. The stale copy is queried without another statistics refresh. Its planner statistics therefore remain those collected from the pre-drift database. The fresh copy receives a database-wide `ANALYZE` after the update. The SQL and current table values are therefore the same in each stale/fresh pair while the planner statistics are different.

4.3. Execution, Measurement, and Censoring

Each query is executed with `EXPLAIN (ANALYZE, BUFFERS, FORMAT JSON, TIMING OFF)`. Setting `TIMING OFF` avoids collecting per-node timing while retaining PostgreSQL's reported execution time, estimated rows, and actual rows. In the depth phase, a condition that completes successfully is executed twice for warm-up and then seven times for measurement. The breadth phase uses one warm-up followed by three measured executions for an uncensored condition. The lower repetition count was chosen to make evaluation of the complete 113-query JOB workload feasible on the hosted runner. For uncensored query-conditions, the reported runtime is the median of the measured executions.

We set the statement timeout to 180,000 ms. A query that reaches this limit is recorded as right-censored because its actual execution time is greater than 180 s and remains unknown. If the warm-up reaches the timeout, further executions for that query-condition are skipped. If a measured execution reaches the limit, the timeout is recorded and the remaining repetitions are not run. When the stale condition is censored but the corresponding fresh condition completes, the stale/fresh runtime ratio is reported as a lower bound. Censored stale/fresh pairs are excluded from the paired inferential test.

For each successful execution, we retain the PostgreSQL JSON plan and extract execution time, planning time, estimated and actual rows, node-level Q-errors, join operators, scan operators, and buffer fields. We also construct a structural plan signature by traversing the plan tree and recording the node type together with the relation name. This signature is coarse because it does not include join predicates, join direction, or several executor properties. Equal signatures therefore indicate similar plan topology rather than identical execution plans. The analysis also stores sums of node-level buffer fields. Since these fields can be cumulative across plan nodes, their sums may count the same activity more than once. We retain them for inspection but do not use the summed values to quantify physical I/O.

4.4. Statistical Analysis

For the depth phase, we calculate stale and fresh query medians at each drift level and compare the paired values using a one-sided Wilcoxon signed-rank test with the alternative that stale runtime is greater than fresh runtime. The depth set contains eight JOB queries and five TPC-H query forms selected for repeated mechanism analysis. These queries are not a random sample of either benchmark. The resulting p values are therefore reported for the selected depth set and are not used to estimate benchmark-wide slowdown prevalence.

For the breadth phase, the main analysis uses the distribution of stale/fresh median-runtime ratios across JOB queries. The slowdown thresholds were fixed at 1.25×, 2×, 10×, and 100×. For the 106 uncensored stale/fresh pairs, we calculate a deterministic 10,000-replicate bootstrap 95% confidence interval for the median ratio. The same 106 pairs are used in a paired one-sided Wilcoxon signed-rank test. Timeout observations are reported separately, and family-level summaries group queries by their numeric JOB family prefix.

We separately measure the duration of the database-wide `ANALYZE` performed in the breadth phase. Its scale is compared with the difference between the capped sums of stale and fresh query medians. Seven stale query-conditions are right-censored at 180 s, so the stale workload sum is not exact. Consequently, this calculation is reported only as a descriptive maintenance comparison rather than an exact workload break-even estimate.

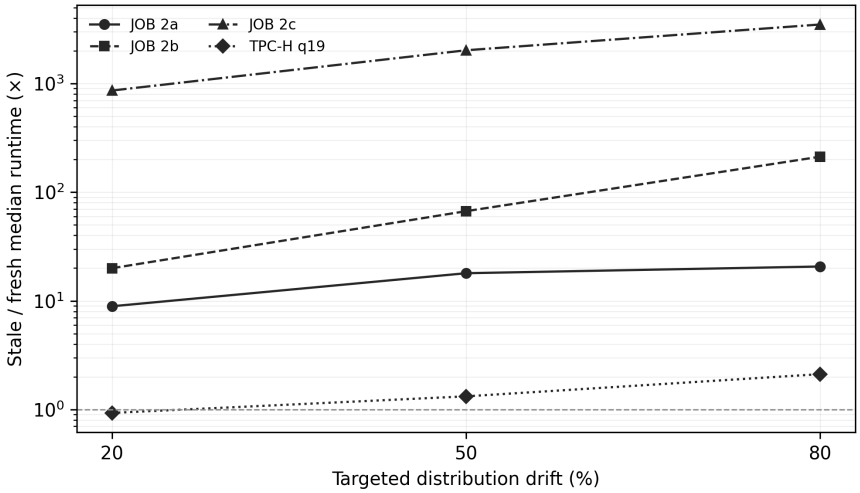

Figure 1: *Runtime-ratio progression for JOB queries.*

Table 3: *Depth-phase runtime progression for selected cases.*

Query	Drift	Stale (ms)	Fresh (ms)	Stale/Fresh
2a	20%	24,366.4	2,733.3	8.91×
2a	50%	82,110.1	4,572.9	17.96×
2a	80%	154,623.7	7,469.8	20.70×
2b	20%	20,997.4	1,051.5	19.97×
2b	50%	56,672.3	847.8	66.85×
2b	80%	95,874.3	452.3	211.99×
2c	20%	18,881.4	21.9	861.81×
2c	50%	51,167.1	25.3	2022.26×
2c	80%	93,219.0	26.7	3490.95×
q19	20%	685.2	736.3	0.93×
q19	50%	1,632.0	1,234.8	1.32×
q19	80%	2,597.3	1,227.8	2.12×

5. Depth Results: Sensitivity to Drift

Table 2 reports the aggregate results of the depth phase. For the eight JOB queries, the median stale/fresh runtime ratio changes only slightly across the three drift levels: 1.011× at 20%, 0.997× at 50%, and 1.045× at 80%. The upper part of the same distribution behaves differently. The p90 of the per-query runtime ratios increases from 272.5× at 20% drift to 653.5× at 50% and 1195.7× at 80%. The five TPC-H forms show smaller changes in this experiment. Their median ratios are 0.931×, 1.018×, and 1.021×, while the corresponding p90 ratios are 0.996×, 1.249×, and 1.825×. The plan-change columns in Table 2 give the percentage of queries whose coarse structural signature differs from the baseline plan.

Figure 1 shows how the selected query ratios change as the targeted drift increases. For JOB 2a, the stale/fresh ratio is 8.91× at 20%, 17.96× at 50%, and 20.70× at 80%. Query 2b changes more sharply, from 19.97× to 66.85× and then to 211.99×. The largest ratios in this depth set occur for query 2c. Its stale execution is 861.81× the fresh execution at 20% drift, 2022.26× at 50%, and 3490.95× at 80%. The five JOB control queries show a different pattern at 80% drift: 3a, 3b, 3c, 6a, and 6b range from 0.982× to 1.080×. Because these control queries and the high-ratio family-2 queries were measured in the same experiment, a uniform slowdown affecting all queries equally does not match the observed pattern.

Table 3 reports the corresponding stale and fresh median runtimes for these JOB cases together with TPC-H q19.

TPC-H q19 also changes with drift, although its ratios remain below those of JOB 2a-2c. At 20% drift, the stale runtime is 685.2 ms compared with 736.3 ms after `ANALYZE`, giving 0.93×. At 50%, the ratio increases to 1.32×, and at 80% it reaches 2.12×. These values provide the TPC-H case shown in Figure 1 without extending the result to the other TPC-H queries.

Table 4: *Breadth results at 80% targeted drift.*

Outcome	Result
Queries analyzed	113
Uncensored stale/fresh pairs	106
Median stale/fresh ratio	1.0059×
Bootstrap 95% CI	0.9916-1.0441
Wilcoxon one-sided p	0.02736
Stale timeouts	7
Fresh timeouts	0
Median stale p90 node Q-error	197.58
Median fresh p90 node Q-error	108.00
`ANALYZE` duration	27.287 s

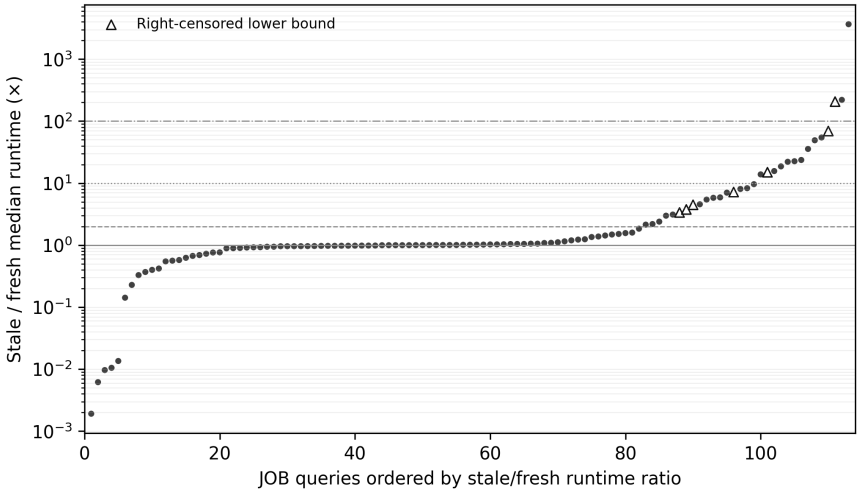

Figure 2: *Stale/fresh runtime ratios for all 113 JOB queries, ordered by magnitude.*

The one-sided Wilcoxon results also change with drift. For the eight JOB queries, the p values are 0.156 at 20%, 0.371 at 50%, and 0.0078 at 80%. For the five TPC-H forms, the corresponding values are 0.969, 0.219, and 0.0313. No depth query timed out in either the stale or fresh condition. These tests apply only to the selected depth queries because this query set was chosen for repeated mechanism analysis. Slowdown prevalence across JOB is evaluated separately using the complete 113-query breadth experiment.

6. Breadth Results: All 113 JOB Queries

The breadth phase evaluates all 113 canonical JOB queries and produces 1003 execution or timeout records. All baseline and fresh query-conditions complete within the 180 s statement limit. Seven stale conditions reach that limit. Their exact runtimes are unknown and are therefore treated as right-censored observations. The median confidence interval and paired Wilcoxon test consequently use the 106 stale/fresh pairs for which both runtimes are uncensored. Table 4 summarizes the main breadth-phase results at 80% targeted drift.

For the 106 uncensored pairs, the median stale/fresh runtime ratio is 1.0059×. Its bootstrap 95% confidence interval is 0.9916-1.0441 and therefore includes 1. The fixed slowdown thresholds give a different view of the same workload. Thirty-nine of the 113 queries (34.51%) are confirmed at or above 1.25×, 31 (27.43%) are at or above 2×, 14 (12.39%) are at or above 10×, and three (2.65%) are at or above 100×. A censored query is included in a threshold count only when its 180 s lower bound already exceeds that threshold. The one-sided Wilcoxon test over the 106 uncensored pairs gives $p = 0.02736$. This result provides evidence of an upward paired runtime shift under stale statistics, although the confidence interval for the median itself still contains 1.

Figure 2 shows the per-query runtime ratios. The largest exact value is obtained for query 2c at 3660.16×, followed by query 2b at 220.29×. Query 17c does not complete within 180 s under stale statistics but completes in 871.8 ms after `ANALYZE`. Its stale/fresh ratio is therefore greater than 206.48×; 206.48× is only the lower bound obtained from the statement cap.

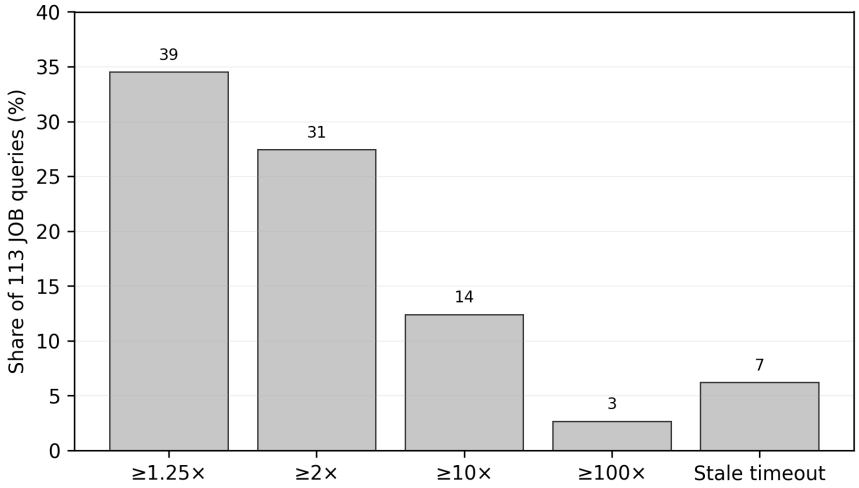

Figure 3: *Confirmed slowdown prevalence at 80% drift.*

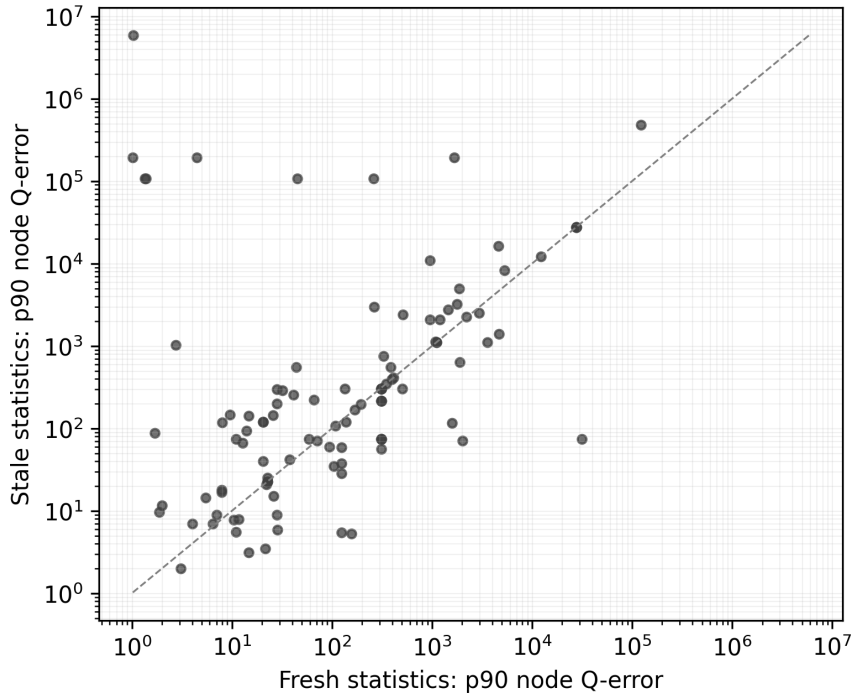

Figure 4: *Query-level p90 node Q-error under stale and fresh statistics.*

The threshold results in Figure 3 are calculated over the complete 113-query JOB workload. In particular, 27.43% of the workload is confirmed at or above 2× and 12.39% is confirmed at or above 10×. These prevalence values come only from the breadth phase; the selected depth queries are not used in these percentages.

The node-level estimates also differ between the two conditions. Across the workload, the median of the query-level p90 node Q-error values is 197.58 under stale statistics and 108.00 after `ANALYZE`. Figure 4 shows that this reduction is not observed for every individual query. This is consistent with prior work showing that Q-error alone does not determine end-to-end plan quality [17, 18]. The plan cases examined in Section 7 are used to determine where large runtime changes coincide with specific estimation and operator changes.

The slowdown is concentrated in a subset of JOB families. Family 2 contains four queries and has a median stale/fresh ratio of 134.72×; all four queries exceed both 2× and 10×, and two exceed 100×. The corresponding family medians are 23.65× for family 26, 17.15× for family 16, and 11.29× for family 32. Family 17 requires a different interpretation because all six stale query-conditions time out. The median of its recorded lower-bound ratios is 11.09×, so its true median stale/fresh ratio is at least 11.09×. Figure 5 reports the eight families with the largest recorded family medians.

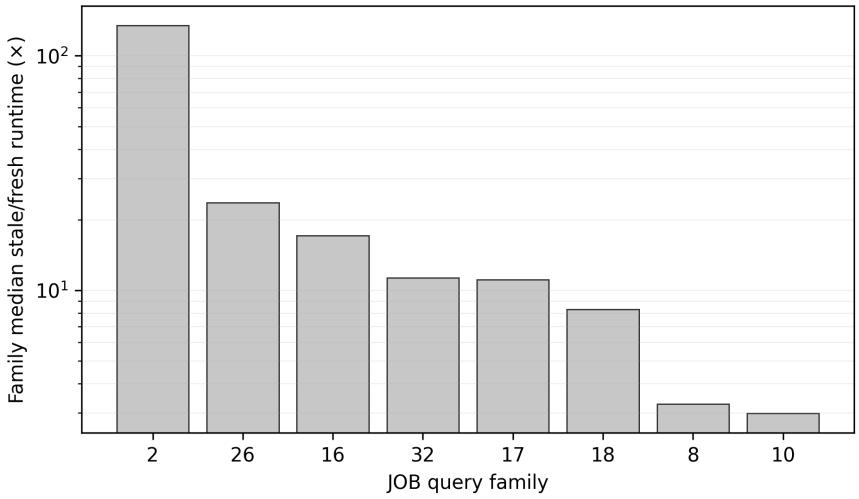

Figure 5: *Eight JOB query families with the largest recorded median stale/fresh ratios.*

7. Plan-Level Mechanisms

7.1. Family 2

Family 2 provides detailed plan evidence because its queries have large stale/fresh ratios and the same family is also evaluated repeatedly in the depth phase. For breadth query 2a, the stale execution uses a chain of nested loops. One nested-loop node is estimated at four rows but returns 23,412,678 rows, corresponding to a Q-error of 5.85 million. Another nested loop is estimated at 19 rows and returns 3,651,454 rows. The median stale runtime is 131.53 s. After `ANALYZE`, the representative plan uses hash joins, its p90 node Q-error is approximately 1.04, and the median runtime is 5.95 s. The resulting stale/fresh ratio is 22.11×.

Query 2b exhibits a related pattern. Its median runtime is 77.81 s with stale statistics and 353 ms after `ANALYZE`, giving a ratio of 220.29×. The p90 node Q-error changes from 192,181.8 in the stale condition to 4.44 in the fresh condition. Query 2c has the largest exact runtime ratio in the breadth experiment. The stale `keyword` scan estimates one row but returns 107,350 rows. A downstream nested loop is estimated at 19 rows and returns 3,651,454 rows. Median runtime is 80.20 s. In the fresh plan, the join order changes and an intermediate nested-loop result reaches zero rows before the later stale-plan work is performed. The fresh median is 21.9 ms, giving an exact ratio of 3660.16×.

Taken together, these plans are consistent with stale cardinality estimates affecting the relative cost assigned to alternative join strategies. The stale plans process intermediate relations that are much larger than their estimates, whereas the fresh plans use different operator topologies in the same query instances. The mechanism claim is therefore based on the combination of estimated and actual row counts, operator changes, and execution time. The stored buffer totals are not used to infer an I/O multiplier because the analysis sums node-level counters that can be cumulative.

7.2. Other Severe Cases

Large runtime ratios also occur outside family 2. Query 26c has a stale median of 124.07 s and a fresh median of 2.28 s, giving 54.50×. One nested-loop node in its stale plan is estimated at 143 rows and returns 88,442,671 rows; the Q-error at that node is greater than 618,000. Query 32b is 22.58× slower under stale statistics and contains a `keyword` estimate of one row compared with 107,350 actual rows. Query 16a has a stale/fresh ratio of 35.63×. Family 17 contains six stale timeouts. For query 17c, the available data support only a lower bound greater than 206.48×. Table 5 brings together eight breadth-phase queries for which the runtime regression can be examined alongside node-level cardinality error and plan structure.

7.3. Plan Change as a Diagnostic

Plan-signature comparisons require uncensored executions because a timed-out condition does not provide the same completed-plan signature as a successful execution. We therefore restrict this analysis to the 106 uncensored stale/fresh pairs. Relative to baseline, the stale coarse signature changes for 34 of these 106 queries (32.08%), whereas the fresh signature changes for 70 of 106 (66.04%). A higher fresh change rate is not itself evidence of better plans. The underlying value distribution has changed, so updated statistics can legitimately cause the optimizer to choose a different topology.

Within the same 106 uncensored pairs, 71 queries have different stale and fresh coarse signatures and 35 have matching signatures. Among the 71 different-signature queries, 18 are at least 2× slower under stale statistics and 10 are at least 10×

Table 5: *Breadth-phase plan cases with large, interpretable regressions.*

Q	Base	Stale	Fresh	Ratio	Stale Q	Fresh Q	Plan observation
	Runtime (ms)				p90 node Q-error		
2c	362.2	80,197.8	21.9	3660.2×	192,182	1,677	Stale nested-loop chain; fresh path reaches zero intermediate rows
2b	369.3	77,810.3	353.2	220.3×	192,182	4.44	Stale nested loops; fresh hash-join topology
17c	13,760.6	> 180, 000	871.8	> 206.5×	censored	20.8	Stale timeout; runtime ratio is a lower bound
26c	3,771.7	124,069.0	2,276.7	54.5×	10,967	948.9	Nested loop: 143 estimated versus 88,442,671 actual rows
2d	525.7	108,946.0	2,216.5	49.2×	192,182	1.02	Fresh execution uses a hash-dominant topology
16a	219.0	29,102.0	816.7	35.6×	107,350	261.3	Keyword scan: 1 estimated versus 107,350 actual rows
32b	119.6	8,299.1	367.6	22.6×	107,350	45.0	Fresh execution uses merge/memoized operators
2a	375.9	131,525.6	5,948.9	22.1×	5,853,170	1.04	Stale nested loops; fresh hash-join topology

slower. Among the 35 matching-signature queries, six exceed 2× and one exceeds 10×. Thus, 10 of the 11 uncensored queries with a confirmed slowdown of at least 10× also have different stale and fresh coarse signatures. The seven stale timeouts are reported separately and are not classified as completed-plan topology changes.

The structural signature remains a coarse diagnostic. It records node type and relation name but omits predicates, join direction, detailed scan properties, and executor state. Two plans can therefore share the same signature while differing in details that affect runtime. Condition order also remains a possible source of timing variation because the stale condition is executed before the fresh condition. For these reasons, the plan-level interpretation relies most heavily on queries such as 2a-2d, 16a, 26c, and 32b, where changes in row estimates, operator topology, and runtime are observed together.

8. Cross-Phase Consistency

Eight JOB queries are executed in both phases at 80% drift. The breadth measurements come from a separate database clone and use a different repetition schedule from the depth phase. Query 2a has a stale/fresh ratio of 20.70× in depth and 22.11× in breadth, corresponding to a 6.8% relative difference. Query 2b changes from 211.99× to 220.29× (3.9%), while query 2c changes from 3490.95× to 3660.16× (4.8%). The five control queries remain around a ratio of 1 in both executions.

Table 6: *Cross-phase consistency for overlapping JOB queries at 80% drift.*

JOB query	Depth ratio	Breadth ratio	Relative difference
2a	20.700×	22.109×	+6.8%
2b	211.986×	220.290×	+3.9%
2c	3490.955×	3660.163×	+4.8%
3a	1.080×	1.056×	−2.2%
3b	1.007×	0.978×	−2.9%
3c	1.005×	0.964×	−4.0%
6a	0.982×	0.962×	−2.0%
6b	1.011×	0.981×	−2.9%

These measurements in Table 6 provide an internal consistency check rather than an external replication. The family-2 ratios differ by 3.9-6.8% between the two executions, whereas the five control-query ratios remain close to 1. Because the breadth values are obtained from a separate execution rather than by reusing the depth measurements, this agreement provides additional evidence that the family-2 behavior is repeatable under the two experimental runs.

9. Discussion

9.1. Stale Statistics as Tail Risk

The breadth experiment gives different results depending on whether the workload is summarized by its median or by its slower queries. The uncensored median is 1.006×, but 31 of 113 queries (27.43%) are at least 2× slower and 14 (12.39%) are at least 10× slower. Seven additional stale query-conditions exceed 180 s. A workload median alone would therefore not reveal these query-specific regressions.

This pattern is consistent with earlier JOB studies in which correlated data produced substantial cardinality-estimation errors [6, 7]. It is also consistent with work showing that Q-error does not map directly to final plan quality [17, 18]. In the present experiment, the clearest large regressions occur when large cardinality underestimates and changes in join topology appear together. Among the 11 uncensored queries with slowdowns of at least 10×, 10 have different stale and

fresh coarse signatures. This association supports the plan-level interpretation, while the matching-signature cases show that topology alone does not account for every runtime difference.

9.2. Implications for Statistics Maintenance

PostgreSQL determines when automatic statistics collection is needed from table modification activity rather than from a direct measure of distribution mismatch [5]. In the breadth experiment, the database-wide `ANALYZE` operation takes 27.287 s. The recorded stale query medians sum to 2327.5 s, compared with 439.1 s in the fresh condition. The stale aggregate includes seven executions capped at the 180 s statement limit, so 2327.5 s is itself a lower bound. These measurements allow the statistics-refresh cost to be compared with query execution time for the tested workload, but they do not provide an exact break-even point or a maintenance threshold that can be generalized beyond this experiment.

The experiment was not designed to determine how often a full `ANALYZE` should be executed. PostgreSQL already provides automatic analysis and configurable statistics targets [4, 5]. The results instead suggest that statistics freshness deserves attention when expensive multi-join queries depend on selective columns whose value distributions have changed but whose stored statistics have not yet been refreshed. Whether such cases justify targeted `ANALYZE` operations or distribution-aware refresh triggers requires a separate evaluation of maintenance cost, query frequency, and workload behavior.

The retained execution plans provide an additional diagnostic use for statistics refresh. When a query regresses after a substantial data update, its plans before and after `ANALYZE` can be compared to determine whether estimated row counts, join operators, or plan topology change after the statistics are refreshed. In the high-ratio cases examined in Section 7, this comparison helps distinguish regressions associated with stale cardinality estimates from runtime differences for which the coarse plan structure remains unchanged. In queries 2a-2d and several other high-ratio cases, those changes occur together with large runtime reductions. Where the estimates and coarse topology remain similar, the present experiment cannot separate statistics effects from other factors such as page-cache state, physical layout, or execution-environment variation.

9.3. Relation to Robust and Learned Optimization

LEO uses cardinalities observed in earlier executions as feedback for later optimization [8], while progressive optimization can reconsider a plan during query execution when observed cardinalities depart from its assumptions [9]. Learned cardinality-estimation work addresses the estimation problem through models of correlated or joint data distributions [11–13]. Distribution shift introduces an additional concern for learned database components, as examined by Zeighami and Shahabi [15], while Roq incorporates uncertainty from a learned cost model into plan selection [16]. Flow-Loss and the benchmark analysis of Han et al. further show why estimation error must be interpreted in relation to the plans it affects [17, 18]. The paired stale/fresh design used here could be adapted in future work to evaluate how these methods behave after a controlled change in the underlying distribution.

PostgreSQL histograms and learned models use different representations, but both depend on information collected or learned from a particular data state. When the deployment distribution changes, that information can lose accuracy. For this reason, a robustness evaluation can report both a typical-query statistic and the proportion of queries that cross predefined slowdown thresholds.

10. Limitations

Following are the limitations of our study.

- All experiments use PostgreSQL 16.4. Planner behavior can change between PostgreSQL releases because changes to cardinality estimation, costing, or optimizer implementation may lead to different plan choices. The runtime ratios reported in this study therefore apply to the tested PostgreSQL version. They should not be assumed to hold for another PostgreSQL release or a different DBMS.
- The experiments run on a GitHub Actions hosted runner rather than dedicated database hardware. The workflow specifies the software environment, but the underlying runner can still be affected by background activity, virtualized storage, and operating-system cache state. These factors can influence absolute execution time. Repeated measurements give multiple observations under the same experimental procedure, but they do not remove variability introduced by the hosted environment.
- Within each drift level, the stale condition is executed before the fresh condition. A forced `CHECKPOINT` is issued after the update phase, and warm-up executions precede the measured runs. These steps separate the update activity from query measurement, but they do not reset the operating-system page cache. Runtime differences are also observed for some stale/fresh pairs with matching coarse plan signatures. Order and cache effects therefore remain possible. The plan-level interpretation gives greater weight to cases in which runtime, estimated row counts, and plan topology change together.

- The JOB drift is produced by deterministic modulo-based updates to three non-key attributes. This construction is useful for creating the same statistical mismatch in paired database copies, but it represents one controlled form of distribution change. Production databases may experience different update patterns, magnitudes, and correlations. The complete 113-query JOB workload is evaluated only at 80% targeted drift. The 20% and 50% settings are evaluated in the selected depth queries only.
- The depth phase uses eight JOB queries and five TPC-H query forms selected for repeated plan analysis. They were not randomly sampled from either benchmark. The slowdown frequencies observed in this 13-query set are therefore not used to estimate workload-wide prevalence. Prevalence is measured separately with all 113 canonical JOB queries at the 80% drift setting.
- The structural signature records the plan-node type and relation name. It does not include join predicates, join direction, detailed scan properties, or several executor settings. Plans with equal signatures can therefore still differ in these properties. A stale condition that times out also does not provide a completed execution-plan signature comparable with a successful fresh execution. For this reason, the stale/fresh plan-signature analysis in Section 7.3 is restricted to the 106 uncensored pairs.
- The analysis code records PostgreSQL buffer fields and sums their values across plan nodes. Some of these node-level counters can be cumulative, so summing them can count the same activity more than once. The recorded buffer values are retained for diagnostic inspection. Their summed values are not used to quantify physical I/O or to support the plan-level interpretation in Section 7.
- The TPC-H component uses scale factor 1 and five representative query forms. The SQL files are hand-coded forms rather than the complete official `qgen` workload, and the PostgreSQL setup uses an experiment-specific index configuration. The TPC-H measurements are used as a synthetic control in the depth phase and are not reported as an official TPC-H benchmark result.
- Seven stale JOB query-conditions reach the 180 s statement limit. Their execution continues beyond the observation boundary, so their exact runtimes are unknown. The corresponding stale/fresh ratios are reported as lower bounds. Workload sums that use the 180 s boundary for these conditions are likewise capped summaries rather than exact stale-runtime totals.
- The breadth experiment records one database-wide `ANALYZE` duration of 27.287 s. This measurement was obtained for the tested JOB database, PostgreSQL configuration, and hosted runner. It is used only to describe the scale of statistics-refresh cost in this experiment and should not be generalized as a PostgreSQL maintenance-time estimate.

11. Reproducibility

The retained depth artifacts contain the raw execution CSV, query-level summaries, statistical-test outputs, table-statistics snapshots, PostgreSQL JSON plans, and generated figures. The depth artifact contains 637 execution/timeout records and 91 plan JSON files. The breadth artifact contains 1003 execution/timeout records and 353 files in total, including the baseline, stale, and fresh outputs for the complete JOB query set.

The breadth workflow pins PostgreSQL to version 16.4 and checks for exactly 113 canonical JOB SQL files before query execution begins. The workflow also records the drift configuration and run settings in an experiment manifest. The May-2013 IMDb snapshot and JOB SQL are obtained from the public benchmark resources associated with Leis et al. [6, 7]. The breadth protocol is stored separately from the result files so that its thresholds, censoring rules, and statistical analysis can be compared with the resulting output.

The planned reproducibility release includes the experiment code, frozen breadth protocol, raw and summarized CSV files, the plan JSONs required for the cases discussed in Section 7, and the figure-generation scripts. The IMDb dataset can be obtained from its public benchmark source rather than redistributed in the repository, provided that the release records the source and loading procedure. The final public repository URL still needs to be added to the Data Availability Statement before submission.

12. Conclusion

We evaluated PostgreSQL 16.4 after controlled changes to the database value distribution while keeping the current data and SQL identical between paired stale and freshly analyzed copies. In the depth experiment, JOB queries 2a-2c become progressively slower relative to their fresh-statistics executions as the targeted drift increases. Query 2c changes from 861.8× at 20% drift to 3491.0× at 80%. The separate breadth experiment repeats the 80% condition over all 113 canonical JOB queries.

The breadth measurements show that the slowdown is concentrated in part of the workload. The uncensored median stale/fresh ratio is 1.006×, while 31 queries are at least 2× slower, 14 are at least 10× slower, and three are at least 100× slower. Seven stale query-conditions exceed the 180 s statement limit, whereas every fresh condition completes. In the plan cases examined in detail, large runtime regressions coincide with substantial cardinality underestimation and changes in join topology. Query 2c provides the largest exact breadth ratio at 3660.16×.

These measurements show why the median alone is insufficient to describe this workload under the tested drift intervention. Many JOB queries remain close to their fresh-statistics runtime, while a smaller group accounts for the large slowdowns and timeouts. Future experiments should randomize stale/fresh execution order and apply the complete JOB workload at lower drift levels. Further evaluation on additional PostgreSQL versions and other DBMSs is also needed. A separate maintenance study could then test targeted statistics refresh and distribution-aware refresh triggers under less severe and more production-like update patterns.

Author Declarations

Author Contributions (CRediT): Sadia Aslam: Conceptualization, Methodology, Software, Validation, Formal Analysis, Visualization. Muhammad Umar Hammed: Writing - Original Draft, Writing - Review & Editing. All authors have read and approved the submitted version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: The study uses the public Join Order Benchmark SQL and May-2013 IMDb snapshot and TPC-H-derived synthetic data as described in the manuscript.

Acknowledgments: The authors have no acknowledgments to declare.

Conflicts of Interest: The authors declare no conflict of interest.

Ethics Statement: Not applicable.

Declaration of Generative AI: During the preparation of this manuscript, the authors used ChatGPT to assist with language editing.

References

[1] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1979, pp. 23-34, doi: 10.1145/582095.582099.

[2] S. Chaudhuri, "An overview of query optimization in relational systems," in *Proc. 17th ACM SIGACT-SIGMOD-SIGART Symp. Principles Database Systems*, 1998, pp. 34-43, doi: 10.1145/275487.275492.

[3] Y. E. Ioannidis and S. Christodoulakis, "On the propagation of errors in the size of join results," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1991, pp. 268-277, doi: 10.1145/115790.115835.

[4] PostgreSQL Global Development Group, "ANALYZE," *PostgreSQL 16 Documentation*. [Online]. Available: https://www.postgresql.org/docs/16/sql-analyze.html

[5] PostgreSQL Global Development Group, "Updating planner statistics," *PostgreSQL 16 Documentation*, Sec. 25.1.3. [Online]. Available: https://www.postgresql.org/docs/16/routine-vacuuming.html

[6] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proc. VLDB Endowment*, vol. 9, no. 3, pp. 204-215, 2015, doi: 10.14778/2850583.2850594.

[7] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann, "Query optimization through the looking glass, and what we found running the Join Order Benchmark," *VLDB J.*, vol. 27, no. 5, pp. 643-668, 2018, doi: 10.1007/s00778-017-0480-7.

[8] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "LEO - DB2's LEarning optimizer," in *Proc. 27th Int. Conf. Very Large Data Bases*, 2001, pp. 19-28.

[9] V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust query processing through progressive optimization," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 659-670, doi: 10.1145/1007568.1007642.

[10] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann, "Cardinality estimation done right: Index-based join sampling," in *Proc. 8th Biennial Conf. Innovative Data Systems Research*, 2017.

[11] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz, and A. Kemper, "Learned cardinalities: Estimating correlated joins with deep learning," in *Proc. 9th Biennial Conf. Innovative Data Systems Research*, 2019.

[12] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica, "Deep unsupervised cardinality estimation," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 279-292, 2019, doi: 10.14778/3368289.3368294.

[13] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig, "DeepDB: Learn from data, not from queries!" *Proc. VLDB Endowment*, vol. 13, no. 7, pp. 992-1005, 2020, doi: 10.14778/3384345.3384349.

[14] PostgreSQL Global Development Group, "CREATE STATISTICS," *PostgreSQL 16 Documentation*. [Online]. Available: https://www.postgresql.org/docs/16/sql-createstatistics.html

[15] S. Zeighami and C. Shahabi, "Theoretical analysis of learned database operations under distribution shift through distribution learnability," in *Proc. 41st Int. Conf. Machine Learning*, PMLR, vol. 235, 2024, pp. 58283-58305.

[16] A. Kamali, V. Kantere, C. Zuzarte, and V. Corvinelli, "Roq: Robust query optimization based on a risk-aware learned cost model," *arXiv preprint arXiv:2401.15210*, 2024.

[17] P. Negi, R. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh, "Flow-Loss: Learning cardinality estimates that matter," *Proc. VLDB Endowment*, vol. 14, no. 11, pp. 2019-2032, 2021, doi: 10.14778/3476249.3476259.

[18] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui, "Cardinality estimation in DBMS: A comprehensive benchmark evaluation," *Proc. VLDB Endowment*, vol. 15, no. 4, pp. 752-765, 2021, doi: 10.14778/3503585.3503586.

[19] PostgreSQL Global Development Group, "PostgreSQL 16.4 release notes," Aug. 8, 2024. [Online]. Available: https://www.postgresql.org/docs/release/16.4/

[20] Transaction Processing Performance Council, *TPC Benchmark H (TPC-H) Standard Specification*, Revision 3.0.1, 2022.