

How Portable is ORM Query Performance? Evidence from Four Database Systems

Mahmoud Nasr, Mohammed El-Ramly, Desoky Abdelqawy
Faculty of Computers and Artificial Intelligence, Cairo University, Giza, Egypt

Abstract—A query that runs efficiently through an object-relational mapper (ORM) on one database may become a bottleneck on another. Such differences are easy to miss when ORM performance is summarized by a single overhead figure. This study examines how reliably performance carries across database systems by comparing Django and SQLAlchemy with handwritten SQL on four systems, with and without secondary indexes. Using TPC-H-derived queries and TPC-C-derived transactions, paired measurements and result-validation checks assess both relative overhead and actual delay. The findings show that small typical analytical overhead can coexist with substantial slowdowns for individual operations. Controlled comparisons identify SQL formulation and bound-versus-literal execution as the dominant components of the largest gaps examined. Transaction results show that a higher percentage overhead need not correspond to a greater delay. Together, these findings provide evidence for deciding which operations warrant closer investigation before deployment or database migration. Excluded formulations, conditional transaction estimates, and measurement variability constrain generalization.

Keywords—Object-relational mapping; database benchmarking; TPC-H; TPC-C; reproducibility; measurement error

I. INTRODUCTION

For a database-backed application, a slow report or transaction can delay the user even when most database operations are fast. Developers therefore need to know which operations require attention before deploying an application or changing its database configuration. An object-relational mapper (ORM) lets them express database operations through objects and query APIs [1], [2], but the same application operation may have different execution costs on another database or with different indexes. The practical problem is deciding where this abstraction adds an acceptable cost and where an operation needs investigation or a SQL rewrite. This study assesses performance portability as the stability of an operation's ORM-to-SQL runtime ratio across database and index configurations.

A single average or median overhead cannot answer that question. It can conceal an expensive operation among many that run close to their handwritten SQL baselines. Replacing ORM code also has a development and maintenance cost, so developers need evidence about individual operations before choosing what to optimize. This motivates a comparison that reports both relative overhead and actual runtime for each database and index configuration.

An ORM may add client work, generate a different SQL expression, or send parameters through a different interface. These changes can affect the time taken by the complete operation. Prior studies examine generated query structure

and redundant data access as sources of performance differences [3], [4]. The main comparison measures the combined cost; controlled follow-up comparisons distinguish incremental client-path work, SQL formulation, and bound-versus-literal execution.

This study examines Django and SQLAlchemy across four database systems and two index configurations. The analytical workload covers 22 operations derived from TPC-H, and the transactional workload covers the five TPC-C transaction types [5], [6]. Together they provide 405 reported comparisons: 325 primary analytical cells and 80 transactional cells, including conditional Order-Status estimates. Analytical timing uses eight paired, counterbalanced blocks; transaction latency uses 24 matched, counterbalanced blocks, subject to the acceptance checks below. The study asks three questions:

- **RQ1:** How does observed analytical overhead vary across queries, database systems, and index configurations?
- **RQ2:** Which of SQL formulation, bound-versus-literal execution, and incremental client-path work dominates the largest observed gaps?
- **RQ3:** How do TPC-C-derived transaction costs vary across operations and configurations, and how do their observed overheads compare with the analytical workload?

The analytical median overhead is 0.86%, but 23 comparisons exceed 100%. This gap shows why a small median gives little assurance about an individual operation. The transaction results add a second finding: a higher percentage increase need not produce a larger delay in milliseconds.

Earlier studies compare Django queries and measure Doctrine across database systems [3], [7]. Building on this work, this study makes three contributions:

- A comparison across operations and configurations. The study reports relative overhead and absolute runtime for Django and SQLAlchemy across four database systems, two index configurations, 22 analytical operations, and five transaction types.
- Analytical estimates with explicit measurement limits. Eight paired, counterbalanced blocks support the analytical comparisons. The study reports coverage, variability, and incomplete comparisons so readers can judge the strength of the evidence.
- Evidence for targeted performance investigation. Controlled SQL comparisons identify the largest measured

TABLE I. POSITIONING AGAINST THE TWO CLOSEST PRIOR ORM–SQL COMPARISONS

Study	Frameworks and databases	Workload	Comparison focus
Colley et al. (2020) [3]	Django; Azure DB	Five query objectives on an oceanographic dataset	Generated versus handwritten SQL; query structure, plan metrics, and execution time
Choina and Skublewska-Paszkowska (2022) [7]	Doctrine; MySQL, PostgreSQL, Oracle	Insert, update, delete, and select operations	Execution time with and without object-relational mapping
This study	Django and SQLAlchemy; four database systems	22 TPC-H-derived operations and five TPC-C-derived transaction types	Overhead and absolute latency under two index configurations; paired analytical timing and explicit coverage limits

components of large gaps, separate plan diagnostics show examples of database work, and transaction summaries distinguish percentage overhead from added milliseconds. These results help developers prioritize operations for further measurement and possible rewriting.

The rest of this paper is organized as follows. Section II reviews related work and identifies the research gap. Section III describes the method. Sections IV and V present the analytical results and examine selected SQL formulations and execution plans. Section VI reports the transaction results. Section VII discusses implications for practice and research, Section VIII addresses threats to validity, and Section IX concludes the paper.

II. RELATED WORK

This section reviews the research most relevant to ORM performance across database systems. It first considers ORM comparisons and data-access design, then examines query behavior and optimizer context, and finally discusses correctness, measurement, and workload scope. Together, these studies establish the context for evaluating performance portability at the level of individual operations.

A. ORM Performance and Data-Access Design

Colley et al. compare ORM-generated and handwritten SQL and examine the resulting query structures [3], [8]. Their 2020 study uses Django and Azure DB. Colley’s doctoral thesis also studies database optimization [9]. Choina and Skublewska-Paszkowska measure Doctrine operations across MySQL, PostgreSQL, and Oracle [7]. Their tests cover insert, update, delete, and select operations. This study examines analytical queries and transactions across systems, asking how their overhead changes with the database and indexes.

Other studies compare ORM technologies within the Java ecosystem. Baran and Muryjas examine analytical processing with four Java ORM technologies [10], while Tudose and Odubasteanu compare JPA, Hibernate, and Spring Data JPA [11]. These comparisons concern different frameworks and implementations, so their rankings do not directly apply to Django and SQLAlchemy.

Table I positions this study against the two closest comparisons discussed above. It contrasts their reported scope without treating differences in experimental design as a ranking of study quality.

Elapsed time is one part of a data-access decision. Alvarez Eraso evaluates maintainability alongside performance, varying query complexity and database size [12]. Armas et al.

examine code length and access time [13], while Bonvoisin et al. investigate performance–energy tradeoffs [14]. Work on data-access architecture [15] and mapper optimization and parallelization [16] also considers how applications use an ORM. This timing study does not assess these other design objectives.

B. Query Behavior and Optimizer Context

The database requests made by an application offer one place to investigate slow operations. Studies of ORM-based software examine redundant data access and other database inefficiencies [4], [17]. Reformulator addresses a specific case, automatically rewriting code affected by the N+1 query problem [18]. HBSniff examines Hibernate mapping and fetching code smells, including N+1-related patterns, with validation on open-source and commercial code [19]. Yang’s research statement places performance within the broader problem of software quality in database-backed applications [20].

Database comparisons provide context for these application-level studies. Martins et al. examine Oracle and PostgreSQL with TPC-H queries under different constraint and index configurations [21]. Gudilin et al. also compare relational database performance [22]. This study uses each database as the setting for an ORM–SQL comparison, measuring each ORM against its own baseline on that system.

Once the database receives a query, its execution plan also affects runtime. Leis et al. study cardinality estimates, cost models, and plan selection [23]. Han et al. distinguish cardinality-estimation accuracy from execution-plan quality [24]. These studies give reasons to inspect plans, but explaining these timings requires the plans used in this experiment.

C. Correctness, Measurement, and Workload Scope

Timing comparisons require implementations that produce equivalent results. CYNTHIA uses differential testing to find errors in ORM systems [25]. This concern applies to this study: a shorter runtime is not evidence of better performance if the operation does different work.

Benchmark design must also make clear what is timed and how variation is handled. Raasveldt et al. discuss pitfalls in database configuration, implementation, and measurement [26]. DBMS-Benchmarker provides repeatable application-level measurements across systems [27], and Kalibera and Jones explain how to account for variation and uncertainty [28]. These concerns inform the protocol; the repetition counts and variability thresholds are choices made for this study.

TABLE II. COVERAGE OF THE 352-CELL ANALYTICAL DESIGN. P: PRIMARY; C: CENSORED; I: INEXPRESSIBLE; N: NUMERIC MISMATCH; E: EXPLORATORY DJANGO Q21 (INCLUDING ONE CENSORED CELL). SYSTEM A DENOTES COMMERCIAL SYSTEM A. TRANSACTION RESULTS CONTAIN 80 CELLS, INCLUDING 16 CONDITIONAL ORDER-STATUS CELLS

System	Plan	P	C	I	N	E
PostgreSQL	88	76	6	0	4	2
MySQL	88	81	5	0	0	2
Oracle	88	84	0	2	0	2
System A	88	84	0	2	0	2

Vogelsgesang et al. document differences between benchmark and production workloads [29]. Van Renen et al. study records from Amazon Redshift, covering repeated queries, changes in workloads, and their wide variation [30]. This evidence limits how far a fixed benchmark suite can represent production applications.

Taken together, the reviewed studies address ORM costs, database configuration, and measurement quality, but do not provide a combined operation-level comparison of Django and SQLAlchemy across four database systems, both index configurations, and both analytical queries and transactions. This leaves a practical gap: developers cannot use those separate comparisons to assess how the same operation's overhead changes across these settings. This work addresses this gap by reporting relative overhead and absolute runtime in each configuration, using paired analytical measurements and recording incomplete comparisons. Its contribution is evidence about where performance varies and which cases need further investigation, within the validation and workload limits stated in this paper.

III. METHOD

This section describes the comparison design, the timing protocol for each workload, and the execution environment. It also explains how overhead is calculated and how timed outputs and transaction state are checked.

A. Design and Coverage

The design comprises 432 ORM–SQL comparisons: 352 analytical and 80 transactional. Each comparison, called a cell, covers one operation, database system, framework, and index configuration. Each implementation of the operation is called a path. The design combines 22 analytical queries and five transaction types with four systems, two frameworks, and two index configurations. Indexed schemas include secondary indexes; non-indexed schemas retain primary keys but omit secondary indexes.

The workloads follow the TPC-H and TPC-C specifications with the deviations described below [5], [6]. Table II shows which comparisons were completed. It also identifies censored cells, which exceeded a time limit, operations unsupported by the evaluated implementation/backend combination under the selected ORM APIs (termed inexpressible), cells rejected for numeric mismatch, and Django Query 21 cells reserved for exploratory analysis. Missing measurements do not mean zero overhead.

Both workloads are derived from TPC specifications; neither is a compliant TPC benchmark or comparable to published TPC-H or TPC-C results [31]. Analytical queries run individually, without refresh streams or the specified power and throughput tests. Transactions run separately without the specified mix, keying times, think times, or ACID audits.

Analytical data are generated at scale factor 1 with a fixed seed through DuckDB's TPC-H extension [32]. The same files supply the four systems, with 6,001,215 rows in the line-item table. The transactional dataset contains ten warehouses and 4,889,141 rows. Each system contributes 20 measured transactional cells, covering both frameworks and both index configurations.

B. Analytical Timing and Overhead

Each ORM implementation is compared with handwritten SQL executed through its framework and driver. Elapsed time includes query construction, database execution, and result handling. The analytical SQL baseline inserts parameter values into the statement, while the ORM passes them as bound parameters. The measured overhead includes this difference as well as client and database work. Each framework is evaluated against its own SQL baseline, so these ratios do not rank framework speed.

Each analytical query is timed in eight blocks. Each block runs four paths: ORM and SQL for each framework. All four paths use the same parameter set within a block. Their order follows a Williams sequence [33]. Two seeded permutations of the four Williams sequences give eight blocks (four sequences repeated twice), balancing execution position and which path precedes another. Parameter set 0 uses qualification values; sets 1–7 use fixed-seed substitutions [5]. Query 18 uses the qualification quantity 300 in set 0 and quantities in the specified substitution range 312–315 in sets 1–7. The qualification value is outside that substitution range but is prescribed for validation by TPC-H [5]. Both paths receive the same value; the reported Q18 estimate covers this mixed input set and is not a compliant TPC-H performance result.

For each cell, the analysis calculates the ORM-to-SQL runtime ratio within each block and takes the median on a logarithmic scale. Converting this ratio to percentage overhead gives:

$$H_c = 100 \left[\exp \left\{ \text{median}_{b=1}^8 \log \frac{t_{\text{ORM},c,b}}{t_{\text{SQL},c,b}} \right\} - 1 \right]. \quad (1)$$

Here, $t_{\text{ORM},c,b}$ and $t_{\text{SQL},c,b}$ are the two times for cell c in block b . Negative overhead means the ORM path completed sooner. Dividing separate ORM and SQL medians would lose the pairing. The study reports those path medians separately to show the actual time in seconds.

Each path runs once as a warmup before its timed run in each block. The ceiling is 900 s per call, with no extended retry in the main experiment. Separate censored-cell checks are reported below. Comparisons that exceed the limit are excluded from numerical overhead summaries but remain in the coverage table. For each database/index campaign, checks cover indexes, refreshed database statistics, resource limits, and

readiness. A short query checks whether execution position affects timing before the main measurements. Warmup and repeated measurements address known benchmarking concerns [26], [28]. These checks cannot exclude drift during a long campaign.

To summarize accepted cells, the analysis takes the median of their log ratios and converts it back to percentage overhead. The 95% confidence intervals use 2,000 bootstrap resamples of the queries retained in each summary (21 for Django and 22 for SQLAlchemy in the primary SF1 analysis). When a query is selected in a resample, all its configurations stay together [34]. This does not make the query suite representative of production workloads.

Exact two-sided Wilcoxon signed-rank tests operate separately for each framework on query-level medians of the cell log ratios [35]. The test assumes independent query-level effects with a distribution symmetric around zero under the null hypothesis. The tests describe this fixed query suite. A query enters with at least one accepted cell; 21 queries enter for Django and 22 for SQLAlchemy. Among included queries, cell counts range from 2 to 8 for Django and 6 to 8 for SQLAlchemy. Supplementary Table S4 lists the counts; queries with fewer configurations have less complete coverage.

The coefficient of variation (CV) is a path's standard deviation divided by its mean, expressed as a percentage. Here it describes variation across parameter sets, not uncertainty in the paired overhead estimate. Pre-pilot criteria require median CV at most 5%, 90th-percentile CV at most 15%, and at most 5% of paths above 25%. The slowest path must also take no more than one fifth of its statement time limit. The authors chose these thresholds and the eight-block budget for this study. Campaigns that fail a criterion remain in the analysis; Section VIII reports their effect.

C. TPC-C-Derived Transaction Protocol

The five operations are New-Order, Payment, Order-Status, Delivery, and Stock-Level [6]. Single-client latency measurements use 24 Williams-balanced blocks with seeded, matched inputs and a separate warmup. For transactional cell c , the overhead uses the same within-block log-ratio calculation as the analytical workload, with the median taken over its accepted blocks. Timing includes the transaction function and commit; input seeding, state capture, validation, and restoration are outside the timer.

Before- and after-images of the affected rows identify committed inserts, updates, and deletes. Each ORM-SQL pair must agree on these changes, declared common output fields, and input keys; fields outside that declared output set are recorded but not compared. The analysis supplement lists the fields for each transaction and their definition in the archived validation code. Clock-generated fields are checked against the call interval rather than required to have identical timestamps. After every call, the starting state is restored and reread, and table counts are checked. Payment history text and SQLAlchemy's Order-Status SQL retrieval use the baseline implementations identified by the source hashes in the execution records.

Django's baseline uses a database cursor; SQLAlchemy's uses Session execution with a text statement. Each ORM is

TABLE III. RECORDED ENVIRONMENT FOR THE MAIN TIMING EXPERIMENTS. MEASUREMENT RECORDS LINK TO ENVIRONMENT INVENTORIES; THE TRANSACTION-VERSION LIMITATION IS STATED IN APPENDIX A

Component	Recorded configuration
Host	Intel Core i7-8550U; Ubuntu 26.04.1
Database / client CPUs	Logical 0–1 / 2–3
PostgreSQL	18.6
MySQL	8.4.11
Oracle native server	23ai Free 23.7.0.25.01
Commercial System A	Version withheld
Memory ceiling	3 GB; swap disabled
Storage device	SK hynix SC311 SATA SSD

compared with its own baseline. The baselines differ, so their overhead percentages cannot establish which framework is faster.

For each transaction type, a framework's summary combines eight cells, one for each system and index configuration. The study reports overhead percentages and absolute times in milliseconds. To summarize overhead, the analysis takes the median log ratio across cells and converts it back to a percentage. Added latency is the difference between the two path medians within a cell:

$$\Delta_c = \text{median}_{b \in A_c} t_{\text{ORM},c,b} - \text{median}_{b \in A_c} t_{\text{SQL},c,b},$$
$$\Delta_{\text{summary}} = \text{median}_{c \in C} \Delta_c. \quad (2)$$

Here, A_c contains the accepted blocks and C the cells in the summary; times are expressed in milliseconds. This is not the median of paired block differences. Differences across the five transaction types describe these operations; they are not confidence intervals.

Separately collected throughput runs use 1, 4, 8, 16, 32, and 50 clients for 60 s after a 10 s warmup. These runs are outside the 432-cell latency design. They execute transaction types separately, so their throughput is not the prescribed TPC-C mix and is not tpmC.

D. Environment and Execution Records

The database services ran one at a time on Ubuntu 26.04.1 LTS. The host used an Intel Core i7-8550U with four physical cores and eight logical CPUs. Database cpusets use logical CPUs 0–1; the client uses 2–3 on the same host. Assigning different logical CPUs does not guarantee separate physical cores. Connections use loopback. Table III summarizes the recorded environment; the database volume is on a SK hynix SC311 SATA SSD.

A systemd cgroup sets a 3 GB memory ceiling and disables swap. Database memory targets are 2 GB where applicable; Oracle's system global area and Commercial System A's configured target are 1.5 GB. Intra-query parallelism is disabled. The client specification is Python 3.12, Django 6.0.8, and SQLAlchemy 2.0.52 [1], [2]. Shared drivers are psycopg2 2.9.12, mysqlclient 2.2.8, oracledb 4.0.2, and pyodbc 5.3.0.

Commercial System A is anonymized because project records identify restrictions on publishing vendor-attributed

TABLE IV. ACCEPTANCE OF THE PRIMARY SF1 ANALYTICAL PAIRS.
EXCLUDED CELLS REMAIN IN THE COVERAGE TABLE

Classification	Pairs
Exact	2,512
Scale-equivalent	88
Total accepted	2,600

benchmark results. Its eligible measurements enter the main analysis with exclusion sensitivity. Withheld identity, version, and plans limit independent reproduction, as detailed in Data Availability.

Each measurement record carries its protocol, environment, dataset, input, and session identifiers, plus source and runner hashes. The environment identifier links the record to inventories containing server versions, settings, indexes, packages, and storage details. Supplementary Note S1 retains the provenance details and remaining metadata limits.

E. Result Equivalence

The full result of each timed analytical call is compared with its framework's SQL baseline outside the timing interval, following differential-testing principles [25]. The predefined acceptance policy preserves named columns, duplicate rows, and required ordering. Only equal-key ties may reorder; midnight date representations and justified numeric representations are classified separately. Decimal scale equivalence is accepted only at the baseline's own declared scale, with at least four fractional digits; money values must agree exactly.

The 325 primary SF1 cells contain 2,600 accepted pairs: 2,512 exact and 88 scale-equivalent. A cell enters the primary analytical summary only when all eight blocks are accepted and no known semantic difference remains. Django Query 21 counts distinct orders rather than qualifying line items; agreement at SF1 does not establish semantic equivalence. Its seven completed SF1 cells and one censored cell are therefore exploratory and excluded from all primary analytical, controlled, and cross-setting summaries. The archive includes these exploratory timings. PostgreSQL SQLAlchemy Queries 8 and 14 in both index configurations are excluded for numeric mismatch. The four inexpressible cells are Django Query 13 on Oracle and Commercial System A, with and without secondary indexes.

The eleven primary censored cells have separate result-agreement evidence: seven completed directly on the second server under an extended ceiling, and four are supported by indexed twins with byte-identical statements and parameters on the same input and data. Twin evidence checks observed agreement of the unchanged formulation; it does not supply the missing non-indexed runtime. Supplementary Note S1 names each case. Transaction acceptance additionally requires the state checks above; rejected Order-Status inputs are reported in Section VI.

Table IV summarizes the accepted analytical pairs.

IV. ANALYTICAL RESULTS

This section addresses RQ1 by examining analytical overhead across queries, systems, and index configurations. It

TABLE V. PRIMARY ANALYTICAL PAIRED OVERHEAD. INTERVALS
RESAMPLE 21 DJANGO OR 22 SQLALCHEMY QUERY CLUSTERS; EACH
RETAINED CELL HAS EIGHT PAIRED BLOCKS. FRAMEWORK-SPECIFIC
SQL BASELINES DIFFER, SO THE ROWS DO NOT RANK FRAMEWORK
SPEED

Framework	Cells	Median (%)	95% interval
Django	158	1.44	[0.53, 4.38]
SQLAlchemy	167	0.60	[0.31, 1.39]

TABLE VI. PRIMARY ANALYTICAL COMPARISONS, GROUPED BY
OVERHEAD H_c . THE GROUPS DO NOT OVERLAP

Overhead range	Comparisons
Below -1%	38
-1% to $+1\%$, inclusive	130
Above $+1\%$ to $+100\%$, inclusive	134
Above $+100\%$	23
Total	325

first describes the distribution of measured overheads, then uses absolute runtimes to show the practical size of selected differences.

A. RQ1: Distribution Across Queries and Configurations

Across 325 primary cells, the pooled median overhead is 0.86%. Framework summaries appear in Table V. The confidence intervals are above zero, with exact query-level Wilcoxon $p = 0.0063$ for Django and $p = 9.1 \times 10^{-6}$ for SQLAlchemy. As defined in Section III, these tests describe the fixed query suite.

The median hides large differences between individual comparisons. Table VI accounts for all 325 primary cells. Each cell belongs to exactly one group. Of the 157 cells above 1% overhead, 134 are at or below 100% and 23 are above it. The $\pm 1\%$ band is descriptive; it is not a statistical equivalence test.

The 23 large-overhead cells include 13 of 168 indexed cells and 10 of 157 non-indexed cells. Missing secondary indexes therefore do not account for every large gap.

In total, 82 cells have negative overhead: 38 are below -1% , and 44 are between -1% and zero. Those 44 are already counted in the group within $\pm 1\%$. Four cells are below -50% . Fig. 1 shows the full range and a closer view around zero.

Commercial System A's medians also depend on the index configuration. With indexes, its median overhead is 27.8% for Django and 10.63% for SQLAlchemy; without indexes, the medians are 10.57% and 15.3%. Excluding Commercial System A leaves a pooled median of 0.51% (95% interval [0.24, 0.89]) and 10 cells above 100%; the finding of a small median with large query-specific gaps therefore persists.

Fig. 2 summarizes each database configuration. Within a single configuration, some operations stay close to SQL while others have large overheads. An engine-level median cannot establish the portability of a particular query.

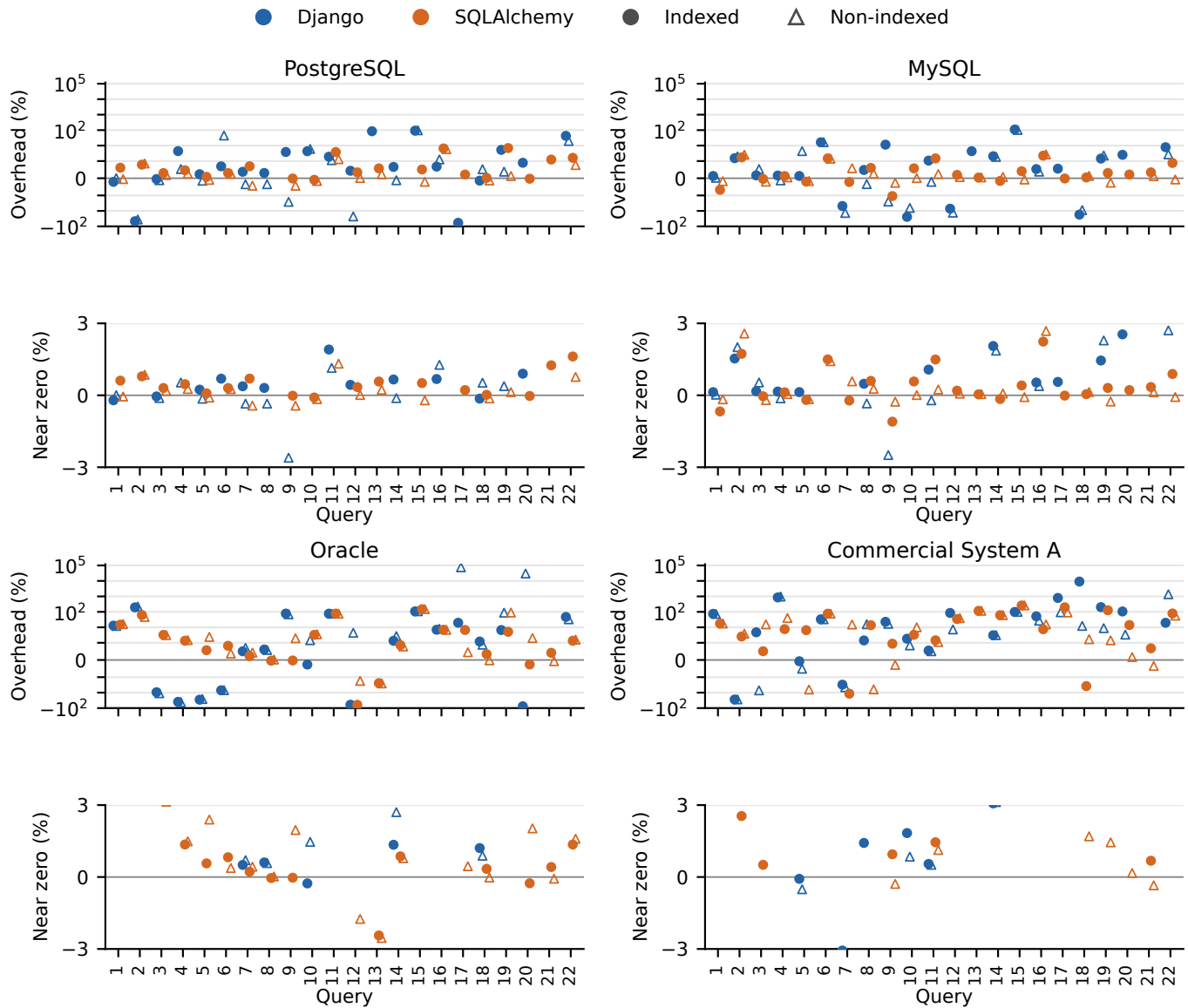


Fig. 1. Primary analytical overhead across the four systems; Django Q21 is excluded. The key shows each framework/configuration combination: blue denotes Django and orange SQLAlchemy; filled circles denote indexed schemas and open triangles non-indexed schemas. The upper panels retain the full range on a symmetric-log axis; lower panels show only the $[-3, 3]\%$ region. Values outside the lower panels remain visible above.

B. Practical Magnitude

Table VII reports paired ratios alongside the median run-time of each path. For Django's indexed Oracle Query 17, the ORM and SQL medians differ by about 8 ms. Without secondary indexes, the ORM median is roughly nine minutes. Absolute times make the cost of this gap easier to judge.

These comparisons locate timing gaps, with the validation evidence and limits described in Section III-E. Section V separates client work, bound-versus-literal execution, and SQL formulation using controlled comparisons.

C. Additional Scale, Client, and Server Checks

Queries 6, 17, 18, 20, and 21 were also tested at scale factor 2, with a physically separate client, and on a second

server. The separate client was an Apple M4 Mac over Wi-Fi. The second-server check used the same Mac with database engines confined to two CPUs and 3 GB in Linux virtualisation; Commercial System A used binary translation. These are subset checks, not repeats of the full query suite.

Table VIII compares matched cells using the ratio of ratios, $(t_{\text{ORM}}/t_{\text{SQL}})_{\text{new}}/(t_{\text{ORM}}/t_{\text{SQL}})_{\text{SF1}}$. Median ratios remain near one, but 14 of 53 SF2 cells and 8 of 63 cells in each of the separate-client and second-server settings fall outside the 0.9–1.1 band. Supplementary Table S3 identifies the largest departures; the supplement provides every matched-cell ratio. These comparisons are conditional on completion in both settings. The SF2 campaign completes 53 cells; 14 Q21 cells stop after a result mismatch, 3 are censored, and 10 host-censored cells are skipped. The separate-client and

TABLE VII. CURATED PRIMARY ANALYTICAL OUTLIERS AND CONFIGURATION CONTRASTS, NOT A REPRESENTATIVE SAMPLE. SECONDS ARE MARGINAL PATH MEDIANS; THE RATIO IS THE PAIRED ESTIMATE AND NEED NOT EQUAL THE QUOTIENT OF THE DISPLAYED SECONDS

Query	System	Indexes	Framework	SQL (s)	ORM (s)	Paired ratio
Q15	PostgreSQL	yes	Django	0.348	0.670	1.92×
Q15	Oracle	yes	Django	0.763	1.617	2.08×
Q15	Oracle	yes	SQLAlchemy	0.765	1.928	2.48×
Q17	PostgreSQL	yes	Django	1.301	0.536	0.41×
Q17	Oracle	yes	Django	0.042	0.050	1.20×
Q17	Oracle	no	Django	0.737	562.778	761.37×
Q20	Oracle	yes	Django	0.850	0.203	0.24×
Q20	Oracle	no	Django	0.914	271.041	301.75×
Q15	Commercial System A	yes	SQLAlchemy	1.079	3.862	3.58×
Q17	Commercial System A	yes	Django	0.102	0.912	8.93×
Q18	Commercial System A	yes	Django	3.982	363.369	91.83×

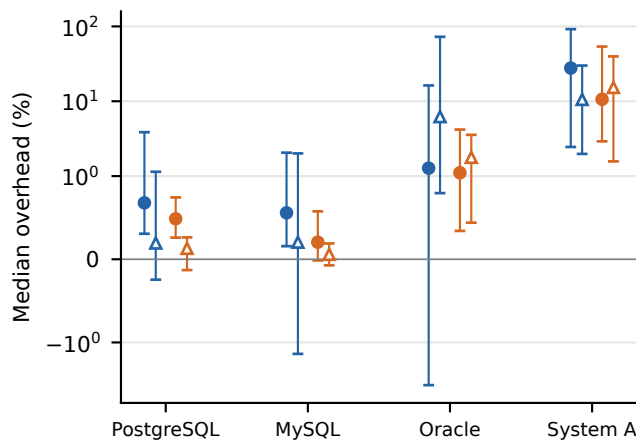


Fig. 2. Primary configuration medians and query-clustered 95% intervals. Blue denotes Django and orange SQLAlchemy. Filled circles denote indexed schemas; open triangles denote non-indexed schemas. The vertical axis is symmetric-log.

second-server checks each complete 70 cells; excluding seven completed Django Q21 cells leaves 63 primary comparisons in each setting. Ten host-censored cells are not repeated.

At SF2, a duplicate within one order exposes Django Query 21's counting difference (Section III-E), changes the top-result boundary, and stops the Q21 comparisons. Their partial timings are excluded. SQLAlchemy Q21 remains eligible where all eight blocks pass.

The first separate-client run of Commercial System A without indexes was affected by server memory state left after the indexed workload. Repeating those cells on a freshly started engine, as in the same-host run, removes most of the difference; the table uses the fresh-engine measurements. The remaining exploratory Django Q21 difference and the hardware limits are discussed in Section VIII.

V. SQL FORMULATION AND EXECUTION PLANS

To address RQ2, the study examines SQL formulations and execution plans, then compares ORM calls, generated SQL,

and handwritten SQL with matched bindings. The controlled comparisons examine selected large gaps and a sample of other cells; their selection and limits are stated below.

A. Query 15: Two Requests and Repeated Expressions

Query 15 finds suppliers with the highest quarterly revenue [5]. The baseline defines revenue in a common table expression and refers to that result twice. Django first calculates the highest revenue, then calculates revenue again to select the suppliers. This sends two statements that both access `LINEITEM`.

The indexed PostgreSQL and Oracle examples show Django ratios close to two. The extra request could contribute to the delay and needs a separate test. SQLAlchemy instead refers twice to an aggregation subquery within one statement; indexed Oracle still shows a 2.48-fold runtime. The number of requests does not establish whether the database calculates a repeated expression once or twice.

B. Query 17: Index Context Changes the Comparison

Query 17 filters line items using a per-part average quantity [5]. The baseline correlates the inner aggregate with `PART`; the Django implementation correlates through the outer `LINEITEM`. Both compute the final sum and division in SQL. The compiled PostgreSQL listing retained with the supporting material is a reconstruction from the archived benchmark source, not a trace of the timed Oracle execution.

On indexed PostgreSQL, the paired Django ratio is 0.41. On Oracle, it is 1.20 with secondary indexes and 761.37 without them. Excluding three blocks overlapping background jobs reduces the latter estimate to 646.13 across five blocks (Section VIII). The Oracle pair allows comparison of index configurations while keeping the database fixed. A comparison between indexed PostgreSQL and non-indexed Oracle would change both factors. Even within Oracle, indexes can affect both SQL formulations.

The different correlation could contribute to the timing gap. Prior optimizer studies give reasons to investigate plan quality [23], [24], but do not identify the plans used in these timed calls. The single-input non-indexed Oracle comparison

TABLE VIII. ADDITIONAL-SETTING CHECKS AFTER EXCLUDING DJANGO Q21. OVERHEAD COLUMNS ARE PERCENTAGES; EXCL. A REMOVES COMMERCIAL SYSTEM A. RATIO-OF-RATIOS COMPARISONS USE THE SAME PRIMARY CELLS COMPLETED IN BOTH SETTINGS. WITHIN 10% MEANS A RATIO OF RATIOS FROM 0.9 TO 1.1

Setting	Cells	Overhead	Excl. A	Ratio of ratios	Within 10%
SF2	53	0.29	0.10	0.993	39/53
Separate client	63	0.79	0.39	0.999	55/63
Second server	63	0.83	0.31	0.997	55/63

in Section V-E records repeated work for one matched input. Isolating the effect of the correlation still requires a controlled change to the SQL.

The Oracle Query 17 and 20 cells pass result-equivalence checks for all eight timed inputs in both index configurations (Section III-E). The four-path comparisons below distinguish formulation and bound-versus-literal execution effects; they do not isolate one syntactic change within a formulation.

C. Commercial System A: Large Query-Specific Costs

Commercial System A contributes 13 of the 23 primary cells above 100% overhead. Django's indexed Query 18 adds 9083%. Table VII shows a curated set of large gaps and configuration contrasts with their times in seconds.

On indexed Query 17, Django adds 793% and SQLAlchemy adds 198%. SQLAlchemy uses a statement with a similar structure to the SQL baseline, but still takes longer. For Query 15, its overhead is 258% with indexes and 263% without them. Database context remains relevant even when the client issues a single statement.

Commercial System A's indexed Django Query 18 is a large outlier in both the controlled comparison and the main analytical experiment, although its ORM times differ on the same server version. The reason is unresolved; Section VIII discusses this difference separately from the within-run bound-versus-literal result.

All eight timed Query 18 input pairs pass the result-equivalence checks. These checks establish agreement for the measured inputs; they do not explain the timing gap.

D. Query 20: Parameter Handling and Index Context

Django's Oracle Query 20 also changes direction between index configurations. With indexes its ORM median is about 0.20 s; without them it is about 271 s. In the indexed campaign, the SQL baselines vary across parameter sets more than the ORM paths. The controlled comparisons below contrast bound and literal execution separately from SQL formulation. For example, PostgreSQL prepared statements may use generic or parameter-specific plans [36]. Kepler likewise evaluates plan selection across parameter bindings using measured execution costs [37]. Such studies motivate binding controls; they do not identify the plan used by a particular Oracle call.

E. Execution-Plan Evidence

Separate single-input diagnostic captures provide concrete examples of database work. Their complete results agreed by named columns and required order. Capture provenance is

summarized in Supplementary Note S1; instrumented times are excluded from the overhead estimates.

For non-indexed Oracle Query 17, the ORM plan scanned `LINEITEM` once for the outer join and restarted an inner full scan 771 times. The baseline scanned the table once and used a window operation on the 6,088 joined rows. Both returned exactly the same scalar result. The plan therefore exposes repeated database work in this diagnostic execution; it does not prove that changing the correlation alone would produce the baseline strategy.

For non-indexed Oracle Query 20, the ORM plan started the inner `LINEITEM` scan 366 times to calculate quantities for candidate part-supplier pairs. The baseline scanned `LINEITEM` once, joined the relevant rows, and grouped the quantities before applying the stock condition. All 186 returned rows agreed exactly, including their order. Repeated scans are therefore visible in both Oracle outliers, although the plans do not isolate the effect of bound parameters from SQL formulation.

For indexed Commercial System A Query 18, both paths returned 57 rows. The ORM-generated plan processed 1.5 million orders through repeated index searches and performed 6,001,215 row lookups in `LINEITEM`. The SQL baseline instead scanned and aggregated `LINEITEM` once before retaining the qualifying orders. For its indexed Query 17, both paths returned the same scalar result, but the ORM-generated plan performed an additional full `LINEITEM` scan; the baseline reused intermediate results from the selected line items. These plans show concrete differences in database work. They do not establish which individual SQL or parameter-handling change caused the optimizer to choose each strategy.

F. Controlled Formulation and Parameterization Comparisons

Four paths hold inputs and indexes fixed: the ORM call (*O*), its generated SQL and bindings executed directly (*G*), handwritten SQL with matched bound values and types (*B*), and handwritten literal SQL (*L*). Eight matched inputs and Williams-balanced order support each comparison; full results are checked across paths. Separate plan diagnostics are described above. *O/G* measures the incremental client-path difference, *G/B* contrasts formulations under matched binding, and *B/L* measures the whole bound-versus-literal execution effect, which can include optimizer plan selection; it does not isolate the computational cost of binding parameters.

Selection includes the union of cells above 100% overhead in an archived initial measurement set or the main SF1 experiment, plus a seeded sample of 16 other cells. Three controlled comparisons were completed before the main SF1 experiment

TABLE IX. CONTROLLED DJANGO COMPARISONS FOR THREE PRINCIPAL CASES. ENTRIES ARE PAIRED COMPONENT OVERHEADS (PERCENT); I AND NI DENOTE INDEXED AND NON-INDEXED. *B/L* INCLUDES ANY PLAN-SELECTION EFFECT; IT IS NOT PARAMETER-BINDING CPU COST. RATIOS ARE SUMMARIZED SEPARATELY ACROSS MATCHED BLOCKS; PRODUCTS OF COMPONENT MEDIANS NEED NOT EQUAL THE MEDIAN TOTAL RATIO

Cell	<i>O/G</i>	<i>G/B</i>	<i>B/L</i>	Largest component
Oracle Q17 (NI)	0.06	76206.20	-0.62	Formulation
Oracle Q20 (NI)	-0.20	18912.72	-1.67	Formulation
Commercial System A Q18 (I)	-0.11	-0.16	6330.62	Bound vs. literal

and are included in this set. Of 42 selected cells, two completed Django Q21 cells are exploratory. Of the 40 eligible cells, 39 complete; Commercial System A's non-indexed Django Query 22 is censored because its bound handwritten path exceeds the limit.

Dominant components are identified by the largest absolute median log ratio. In the 22 primary comparisons whose four-path overhead exceeds 100%, SQL formulation dominates in 15 and bound-versus-literal execution in 7. The absolute *O/G* overhead is at most 4.8%; across all 39 completed primary cells its median is 0.20%. These results locate the major components of the measured gaps, rather than attributing every delay to Python work. Table IX reports Oracle Queries 17 and 20 without secondary indexes and Commercial System A Query 18 with indexes. Supplementary Table S2 gives the four path medians and directly calculated paired *O/L* ratios for these cases. A formulation comparison can change several SQL features, so it does not establish the effect of one rewrite in isolation. In Commercial System A Q18, the 6330.62% *B/L* component identifies a large parameterization-associated execution difference. The separate diagnostic plans compare ORM-generated and baseline SQL, not the matched *B* and *L* calls; they cannot establish its specific optimizer mechanism.

VI. TPC-C-DERIVED TRANSACTION RESULTS

This section addresses RQ3 by comparing transaction overhead in percentages and milliseconds. It then examines the limits of comparisons with the analytical workload and reports failures observed in the separate throughput runs.

A. RQ3: Transaction Type and Observed Overhead

Transaction costs differ when measured as percentages and as milliseconds (Fig. 3). Payment has a higher pooled median percentage overhead than New-Order (107.3% versus 93.0%), but a smaller median additional delay (7.9 versus 22.1 ms). A workload average would conceal these differences.

All 80 planned transactional comparisons were measured across four systems, two frameworks, and two index configurations. Median overhead is 136.1% for Django and 67.2% for SQLAlchemy; the pooled median is 104.7%, including conditional Order-Status estimates. Excluding Commercial System A gives 107.5% across 60 cells. Table X reports the five transaction types with their SQL and ORM latencies and path CVs. Fig. 4 retains every system and configuration.

Each cell uses 24 Williams-balanced blocks with seeded, matched inputs. Database state is restored and verified after every call. Of 1,920 planned blocks, 1,868 were accepted. The 52 rejected blocks all belong to Order-Status (13.5% of its 384

planned blocks): when the customer has no order, the ORM implementations return fewer fields than the SQL baselines. Rejections occur in all 16 Order-Status cells, which retain 18–23 accepted blocks each; their estimates apply only to accepted inputs. Excluding all 16 Order-Status cells gives a pooled median of 109.9% across 64 cells (150.0% for Django and 73.2% for SQLAlchemy), with all 1,536 blocks accepted. The large transaction overhead therefore persists without Order-Status, although this sensitivity covers four transaction types rather than the full five-type set.

These remain descriptive estimates. Across 160 paths, median CV is 5.7%, the 90th percentile is 25.5%, and 12.5% of paths exceed 25% CV. These values exceed the analytical criteria of 5%, 15%, and 5%, respectively. The CVs include variation between seeded inputs. The medians do not establish production predictions or a reliable ordering of frameworks. This limitation applies to every transaction table and figure below.

Delivery has the largest median percentage overhead and additional delay for both frameworks. Across the eight configurations, median additional Delivery latency is 65.48 ms for Django and 89.43 ms for SQLAlchemy. Order-Status has the smallest median overhead for Django; Stock-Level has the smallest for SQLAlchemy.

B. The Analytical–Transactional Contrast

The transaction medians are larger than the analytical medians. The protocols also differ in repetition count and operation structure, so this gap cannot be attributed to workload type alone. Each framework uses its own SQL baseline. These differences prevent a fair framework ranking, in percentages or milliseconds.

Transactions issue multiple SQL statements. More statements could add latency, but transaction type also changes the work done by the database. A controlled test would change the statement count while keeping the rest of the transaction fixed. It would also use matching SQL baselines. These data cannot assign a fixed ORM cost to each statement.

C. Throughput and Connection Failures

In the separate throughput experiment, at 50 clients, 20 of 160 paths record aborts, with a maximum abort rate of 70.5%. The logs identify errors when Django paths open connections to PostgreSQL and Oracle. Their throughput must be read alongside these failures. These throughput measurements describe separately executed transaction types; combining their rates does not yield tpmC or estimate an application's capacity.

TABLE X. TPC-C-DERIVED TRANSACTION LATENCY ACROSS FOUR SYSTEMS AND TWO INDEX CONFIGURATIONS. EACH TRANSACTION/Framework SUMMARY CONTAINS EIGHT CELLS. EACH LATENCY IS FOLLOWED BY THE MEDIAN PATH CV (PERCENT) IN PARENTHESES. CELLS USE 24 MATCHED BLOCKS, WITH 18–23 ACCEPTED FOR ORDER-STATUS. MILLISECONDS ARE MEDIANS OF CELL LATENCIES; OVERHEAD IS THE BACK-TRANSFORMED MEDIAN OF WITHIN-CELL LOG RATIOS AND NEED NOT EQUAL THE RATIO OF THE DISPLAYED MILLISECONDS. EACH Framework USES ITS OWN SQL BASELINE, SO THE ROWS DO NOT RANK Framework SPEED. TRANSACTION VARIABILITY EXCEEDS THE ANALYTICAL CRITERIA

Transaction	SQL ms (CV)	Django			SQLAlchemy		
		ORM ms (CV)	Overhead		ORM ms (CV)	Overhead	
New-Order	19.5 (22.1)	42.6 (25.4)	+135.8%		26.0 (23.5)	42.5 (25.6)	+61.1%
Payment	6.6 (5.5)	14.9 (4.4)	+151.1%		8.0 (6.0)	14.1 (5.3)	+90.9%
Order-Status [†]	4.0 (5.0)	7.6 (2.0)	+99.6%		4.4 (4.5)	6.7 (4.5)	+50.3%
Delivery	33.8 (4.2)	99.9 (2.8)	+228.0%		49.3 (3.2)	138.7 (4.1)	+162.4%
Stock-Level	5.8 (4.8)	11.9 (4.0)	+105.6%		8.0 (6.2)	9.9 (4.0)	+23.7%

[†] Accepted inputs only; 52 of 384 planned Order-Status blocks (13.5%) were rejected.

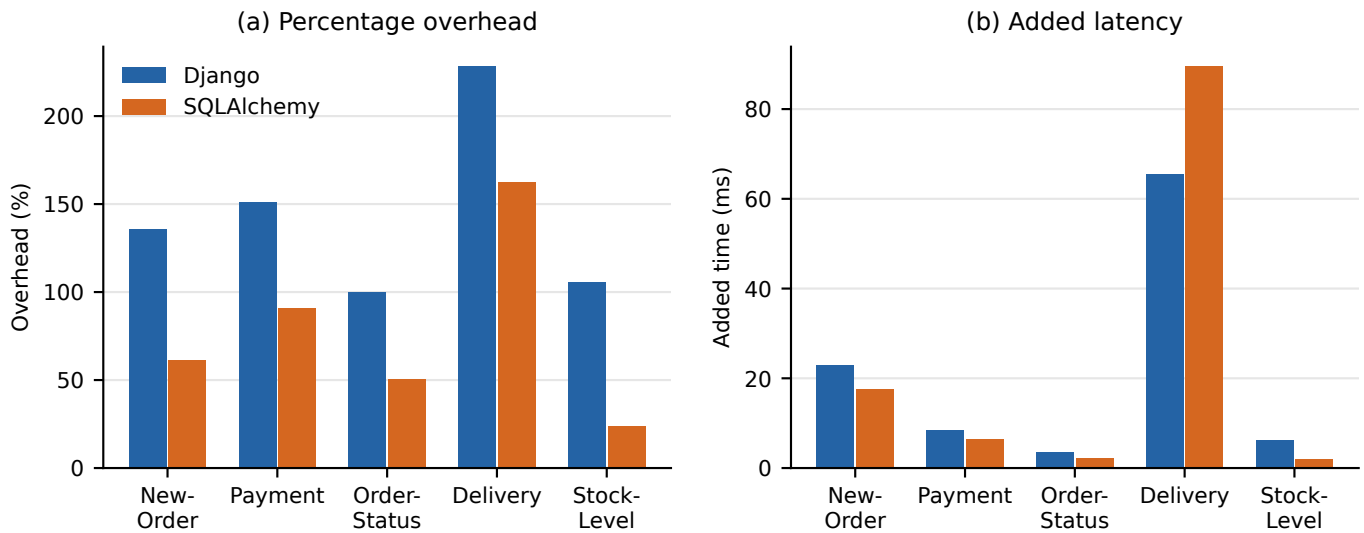


Fig. 3. Transaction overhead expressed as (a) Percentages and (b) Added milliseconds for all five TPC-C-derived types. Each bar summarizes eight database/index configurations for one framework. Percentages are back-transformed medians of within-cell log ratios; milliseconds are medians across cells of the differences between each cell's ORM and SQL path medians (Eq. 2), not differences between pooled medians. Each framework uses its own SQL baseline, so the bars do not rank framework speed. Transaction variability exceeds the analytical criteria; these summaries are descriptive. Order-Status uses accepted inputs only.

VII. IMPLICATIONS FOR PRACTICE AND RESEARCH

This section brings the analytical and transaction findings together and explains how they can inform optimization decisions and further experiments. The recommendations below follow from the observed timing differences; their effectiveness in production has not been tested.

A. What the Results Establish

Small overall overhead can hide a costly operation. The overall analytical median is 0.86%, yet 23 comparisons exceed 100% overhead. Oracle Query 17 shows why this matters: Django's paired ratio changes from 1.20 to 761.37 (646.13 after excluding overlapping blocks) when secondary indexes are absent. Commercial System A contributes 13 of those 23 large-overhead cells. Acceptable performance in one configuration gives no assurance for the same operation in another.

The TPC-C results show that the choice of metric can change optimization priorities. Payment has higher pooled

percentage overhead than New-Order (107.3% versus 93.0%), but a smaller added delay (7.9 versus 22.1 ms). Delivery has the largest median increase in both measures. Protocol differences limit comparisons with the analytical workload, as Section VI explains.

B. What Software Teams Can Use

A change to the database, ORM code, or indexes is a reason to recheck important operations. Teams can compare them with direct SQL in the intended environment, using representative parameters. Both paths should retrieve and process results in the same way. Query 17 shows why an index change needs this check even when the application code stays fixed.

In 130 primary analytical comparisons, the median paired overhead is within $\pm 1\%$. Those timings alone give little reason to replace the ORM code. A large gap calls for checking equivalent results, then inspecting the generated SQL and execution plans. Any proposed rewrite should be measured against

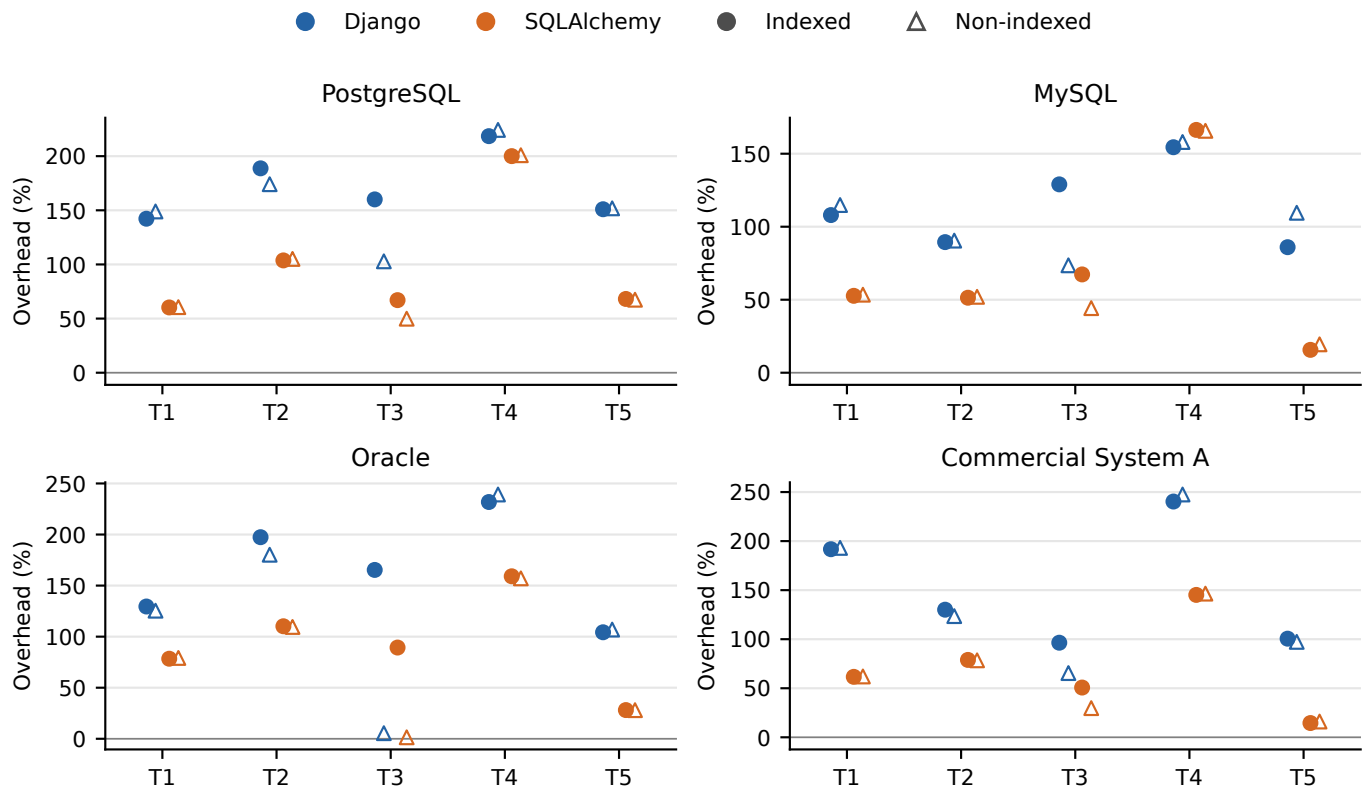


Fig. 4. TPC-C-derived latency overhead for every measured cell. T1: New-Order; T2: Payment; T3: Order-Status; T4: Delivery; T5: Stock-Level. The key shows each framework/configuration combination: blue denotes Django and orange SQLAlchemy; filled circles denote indexed schemas and open triangles non-indexed schemas. Each percentage uses its framework's own SQL baseline. T3 estimates are conditional on accepted inputs.

the original path before adoption. These results motivate this workflow, but its benefit has not been measured.

Choosing what to optimize requires the actual delay and how often the application runs the operation. The benchmark does not measure production request frequencies or latency budgets, so teams must supply those inputs. A large percentage overhead on a rare, short operation may deserve less attention than a smaller overhead on a frequent, slow one. Teams must also consider maintainability when deciding whether to replace ORM code [12].

C. What the Research Community Gains

The study connects 22 analytical operations and five transaction types within one four-system comparison. It reports 325 primary analytical and 80 transactional measurements, with paired input blocks for both workloads. The coverage record also identifies 11 primary censored, four inexpressible, four numeric-mismatch, and eight exploratory Django Q21 analytical cells, so readers can see which observations are missing from the summaries.

The individual cases give researchers specific questions to test. For Query 15, how much does the repeated request cost? For Query 17, how do the two subquery correlations perform with the same indexes? The controlled comparisons in Section V separate the main components of the selected gaps. Further tests can change one SQL feature at a time while

preserving results and bindings. Reporting each operation's time and overhead by database and index configuration would make these comparisons possible in other studies.

VIII. THREATS TO VALIDITY

Three issues limit this study: transaction measurements vary more than planned, some query comparisons are excluded or implementation-specific, and the additional settings cover only a query subset. These limits affect the precision of the overhead estimates and how far they apply to other applications.

A. Variability and Sensitivity

The pooled SF1 analytical paths meet all three CV criteria; transactions exceed them under the 24-block protocol (Table XI). SF2 and the second server meet the median-CV criterion but exceed the other two. Pooled compliance does not imply compliance in every database/index campaign: Commercial System A's indexed and non-indexed SF1 campaigns have 90th-percentile CVs of 20.8% and 19.5%, respectively, above the 15% threshold. Supplementary Table S1 reports path counts and exceeded CV criteria for every campaign. Long-running paths also exceed the predefined target of one fifth of the statement limit.

Tables XII and XIII report exclusions of systems, queries, and acceptance classes. The finding persists without Com-

TABLE XI. PATH VARIABILITY FOR RETAINED CELLS. ALL ENTRIES ARE PERCENTAGES. ANALYTICAL ROWS EXCLUDE PARTIAL CELLS AND DJANGO Q21. THE CV THRESHOLDS ARE 5%, 15%, AND 5%, RESPECTIVELY; CV INCLUDES VARIATION ACROSS SEEDED INPUTS

Dataset	Median CV	90th pct.	CV > 25%
SF1 analytical	1.2	8.3	1.8
SF2 analytical	3.4	26.2	10.4
Separate client	4.1	14.6	4.0
Second server	4.2	15.9	5.6
Transactions	5.7	25.5	12.5

TABLE XII. SENSITIVITY OF PRIMARY SF1 ANALYTICAL MEDIAN OVERHEAD (PERCENT) WITH QUERY-CLUSTERED 95% INTERVALS. DJANGO Q21 IS EXCLUDED FROM EVERY ROW

Subset	Django	95% interval	<i>n</i>	SQLAlchemy	95% interval	<i>n</i>
Primary cells	1.44	[0.53, 4.38]	158	0.60	[0.31, 1.39]	167
Exclude Oracle	1.21	[0.46, 4.10]	118	0.51	[0.22, 1.01]	123
Exclude Commercial System A	0.69	[0.41, 2.29]	118	0.34	[0.16, 0.77]	123
Exclude Q15	1.17	[0.51, 3.10]	150	0.60	[0.30, 1.36]	159
Exclude Q18	1.50	[0.53, 5.97]	150	0.70	[0.34, 1.45]	159
Exclude overhead above 100%	0.89	[0.48, 2.71]	143	0.57	[0.27, 1.32]	159
Indexed only	1.44	[0.52, 4.84]	82	0.69	[0.40, 1.47]	86
Non-indexed only	1.37	[0.53, 5.27]	76	0.43	[0.13, 1.38]	81

TABLE XIII. POOLED SF1 SENSITIVITY. DJANGO Q21 ENTERS ONLY THE LABELLED EXPLORATORY ROW; NUMERIC MISMATCHES ENTER ONLY THEIR LABELLED SENSITIVITY. THE LAST ROW IS A WORST-RANK SCENARIO, NOT IMPUTED RUNTIMES, AND EXCLUDES THE CENSORED DJANGO Q21 CELL

Subset	<i>n</i>	Median (%)	> 100%
Primary SF1	325	0.86	23
Exclude Commercial System A	241	0.51	10
Exclude Oracle	241	0.66	15
Include Django Q21 (exploratory)	332	0.85	25
Exclude scale-equivalent cells	314	0.81	23
Include numeric mismatches (sensitivity only)	329	0.85	23
Censored cells above every primary cell	336	1.01	34

mercial System A; Section IV gives the estimates and Data Availability states its reproduction limits.

System timers remained enabled. Excluding blocks overlapping recorded background jobs leaves the primary pooled medians unchanged. For non-indexed Oracle Django Query 17, however, the paired runtime ratio falls from 761.37 across eight blocks to 646.13 after excluding three overlapping blocks. This is the same outlier discussed in Section V. The exclusion changes the input subset and does not establish that background jobs caused the difference or that the five-block estimate is unbiased. Oracle's slice limits were verified throughout; its intended per-process CPU cross-check did not run because of a case mismatch in the process name.

B. Censoring and Equivalence

The measured-only summaries omit timeouts, inexpressible operations, and rejected pairs. For matched non-indexed Django Query 13 inputs, the measured SQL path and censored ORM path give median ratio lower bounds of 475.1 on PostgreSQL and 219.1 on MySQL. When both paths are censored, the time limit alone does not determine the ratio. Placing all

11 primary censored cells above every primary measured cell gives a worst-rank median of 1.01% over 336 cells, with 34 above 100% in that scenario. These are sensitivity bounds, not invented runtimes.

Scale-equivalent results agree at the declared baseline scale but need not match at every extra digit. The four numeric-mismatch cells stay excluded. Including the seven completed, exploratory Django Query 21 SF1 cells gives 0.85% across 332 cells, with 25 above 100% (Table XIII); the qualitative finding is unchanged. All partial SF2 timings remain excluded. Order-Status estimates exclude no-order inputs that fail output agreement. The sensitivity excluding that transaction type is reported in Section VI; the validation establishes agreement on committed changes and declared common fields, not every returned field. Censored-cell checks on another server or an indexed twin establish result evidence under those settings, not completion of the censored primary-host calls.

C. Scope and Attribution

The full grid uses one primary host and SF1 data. SF2, a separate Wi-Fi client, and a second server broaden the evidence

for a subset; they do not establish production portability [29], [30]. The second server changes processor architecture and virtualisation, and Commercial System A uses binary translation. These factors prevent assigning its timing changes to hardware alone. Non-indexed schemas retain only primary keys and are a stress configuration, not a recommended production design.

The controlled comparisons distinguish formulation, bound-versus-literal execution, and the incremental ORM client path for the selected cells. They do not isolate one SQL rewrite, cover every input distribution, or establish a framework ranking. Commercial System A's indexed Django Query 18 differs between the controlled comparison and the main analytical experiment even on the same server version; the cause remains unidentified. Its exploratory non-indexed Django Query 21 has a smaller ratio in the fresh-engine separate-client experiment than in the same-host run. Those differences remain reported observations rather than explained effects.

IX. CONCLUSION

This study compared Django and SQLAlchemy with hand-written SQL across four database systems and two index configurations. The workloads used 22 TPC-H-derived analytical queries and five TPC-C-derived transaction types. The experiments included result and transaction-state checks, controlled SQL comparisons, and subset checks at another scale, with a separate client, and on a second server.

The primary analytical median overhead was 0.86%, but 23 of 325 comparisons took more than twice as long as their SQL baselines. In the controlled comparisons, SQL formulation and bound-versus-literal execution were the main components of the large gaps studied. The 80 transaction comparisons, including conditional Order-Status estimates, showed that a higher percentage overhead does not necessarily add more milliseconds.

Transaction variability, conditional inputs, excluded formulations, and limited workload coverage constrain generalization. Framework-specific SQL baselines prevent a fair ranking of Django and SQLAlchemy. Future work can test individual SQL rewrites and production workloads beyond the present scope.

DECLARATIONS

Declaration on Generative AI

The authors used AI-assisted tools (Anthropic's Claude and OpenAI's ChatGPT and Codex) for programming support in the benchmark and analysis scripts, for feedback on drafts, and for editing and organising the manuscript. The authors designed the study, reviewed and tested all tool-assisted code, checked query results with the procedure in Section III-E, interpreted the results, and verified the manuscript against the experimental evidence. The authors take full responsibility for the content. No AI tool is an author of this work or is cited as a source.

Data Availability

The experimental records and supporting materials are available at <https://doi.org/10.5281/zenodo.22649752> (release

v2.0.0, commit 63392cfcd10f (12 October 2026), version DOI <https://doi.org/10.5281/zenodo.23311730>). The release retains per-call results and timings, acceptance decisions, protocols, source hashes, environment inventories, and available execution-plan evidence. The identifiers described in Section III bind measurements to those records. Commercial System A remains anonymized; its identity, version, and execution plans are withheld. Its released timings and decisions support reanalysis, but the information required for independent execution is disclosed only for the three identified engines, subject to the documented environment constraints. The Supplementary Materials, Tables S1–S6 and Note S1, are available at <https://doi.org/10.5281/zenodo.23468354>, separately from the experimental archive. This deposit also contains the analysis supplement (`analysis-supplement.zip`, 13 October 2026), which provides the exclusion policy and executable analysis and figure recipes. It reproduces the primary summaries reported in this paper, excluding Django Q21, directly from the public release; the archived release retains its earlier summaries.

Conflict of Interest

The authors declare no conflict of interest. This research was conducted without external funding. The authors declare no conflict of interest. This research was conducted at Cairo University without external funding.

Author Contributions

Withheld for double-blind review. Mahmoud Nasr: Conceptualisation, methodology, software, investigation, data curation, writing of the original draft, visualisation. Mohammed El-Ramly: Supervision, methodology, review and editing. Desoky Abdelqawy: Supervision, review and editing.

REFERENCES

- [1] M. Bayer, "SQLAlchemy," in *The Architecture of Open Source Applications, Volume II: Structure, Scale, and a Few More Fearless Hacks*, A. Brown and G. Wilson, Eds. lulu.com, 2012. [Online]. Available: <https://aosabook.org/en/v2/sqlalchemy.html>
- [2] Django Software Foundation, "Django documentation: Making queries," <https://docs.djangoproject.com/en/6.0/topics/db/queries/>, 2026, Django 6.0, the version measured. Accessed 2026-09-07.
- [3] D. Colley, C. Stanier, and M. Asaduzzaman, "Investigating the effects of object-relational impedance mismatch on the efficiency of object-relational mapping frameworks," *Journal of Database Management*, vol. 31, no. 4, pp. 1–23, 2020.
- [4] T.-H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks," *IEEE Transactions on Software Engineering*, vol. 42, no. 12, pp. 1148–1161, Dec. 2016.
- [5] Transaction Processing Performance Council, "TPC benchmark H (decision support) standard specification," https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.1.pdf, 2022, revision 3.0.1. Accessed 2026-09-09.
- [6] —, "TPC benchmark C standard specification," https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-c_v5.11.0.pdf, 2010, revision 5.11. Accessed 2026-09-09.
- [7] M. Choina and M. Skublewska-Paszkowska, "Performance analysis of relational databases MySQL, PostgreSQL and Oracle using Doctrine libraries," *Journal of Computer Sciences Institute*, vol. 24, pp. 250–257, Sep. 2022.

- [8] D. Colley, C. Stanier, and M. Asaduzzaman, "The impact of object-relational mapping frameworks on relational query performance," in *2018 International Conference on Computing, Electronics & Communications Engineering (icCECE)*, Aug. 2018, pp. 47–52.
- [9] D. Colley, "Development of a dynamic design framework for relational database performance optimisation," Ph.D. dissertation, Staffordshire University, 2021. [Online]. Available: <https://eprints.staffs.ac.uk/7402/>
- [10] J. Baran and P. Muryjas, "The analysis of Java ORM frameworks performance in terms of analytical data processing," *Journal of Computer Sciences Institute*, vol. 27, pp. 178–185, Jun. 2023.
- [11] C. Tudose and C. Odubăşteanu, "Object-relational mapping using JPA, Hibernate and Spring Data JPA," in *2021 23rd International Conference on Control Systems and Computer Science (CSCS)*, May 2021, pp. 424–431.
- [12] D. A. Alvarez Eraso, "A framework for evaluating maintainability and performance of object-relational-mapping tools in web application frameworks," Master's thesis, Universidad Nacional de Colombia, May 2017. [Online]. Available: <https://repositorio.unal.edu.co/handle/unal/59316>
- [13] J. Armas, P. Navas, T. Mayorga, P. Rengifo, and B. Arévalo, "Optimization of code lines and time of access to information through object-relational mapping (ORM) using alternative tools of connection to database management systems (DBMS)," in *2017 2nd International Conference on System Reliability and Safety (ICSRS)*, Dec. 2017, pp. 500–504.
- [14] A. Bonvoisin, C. Quinton, and R. Rouvoy, "Understanding the performance-energy tradeoffs of object-relational mapping frameworks," in *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, Mar. 2024, pp. 626–636.
- [15] F. Majerik and M. Borkovcova, "Design of data access architecture using ORM framework," in *2023 34th Conference of Open Innovations Association (FRUCT)*, Nov. 2023, pp. 93–99.
- [16] F. Jovanov, V. Zdravski, M. Gusev, and M. Kostoska, "Optimization and parallelization of object-relational mappers," in *2023 46th MIPRO ICT and Electronics Convention (MIPRO)*, May 2023, pp. 1678–1683.
- [17] C. Yan, A. Cheung, J. Yang, and S. Lu, "Understanding database performance inefficiencies in real-world web applications," in *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. New York, NY, USA: Association for Computing Machinery, Nov. 2017, pp. 1299–1308.
- [18] A. Turcotte, M. W. Aldrich, and F. Tip, "reformulator: Automated refactoring of the N+1 problem in database-backed applications," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22)*. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 1–12.
- [19] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "HBSniff: A static analysis tool for Java Hibernate object-relational mapping code smell detection," *Science of Computer Programming*, vol. 217, p. 102778, 2022.
- [20] J. Yang, "Improving performance and quality of database-backed software," in *Proceedings Companion of the 2019 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion 2019)*. New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 23–25.
- [21] P. Martins, P. Tomé, C. Wanzeller, F. Sá, and M. Abbasi, "Comparing Oracle and PostgreSQL, performance and optimization," in *Trends and Applications in Information Systems and Technologies*, Á. Rocha, H. Adeli, G. Dzemyda, F. Moreira, and A. M. Ramalho Correia, Eds. Cham: Springer International Publishing, 2021, pp. 481–490.
- [22] D. S. Gudilin, A. E. Zvonarev, B. S. Goryachkin, and D. A. Lychagin, "Relational database performance comparison," in *2023 5th International Youth Conference on Radio Electronics, Electrical and Power Engineering (REEPE)*, Mar. 2023, pp. 1–5.
- [23] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann, "How good are query optimizers, really?" *Proceedings of the VLDB Endowment*, vol. 9, no. 3, pp. 204–215, 2015.
- [24] 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," *Proceedings of the VLDB Endowment*, vol. 15, no. 4, pp. 752–765, 2021.
- [25] T. Sotiropoulos, S. Chaliasos, V. Atlidakis, D. Mitropoulos, and D. Spinellis, "Data-oriented differential testing of object-relational mapping systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, May 2021, pp. 1535–1547.
- [26] M. Raasveldt, P. Holanda, T. Gubner, and H. Mühleisen, "Fair benchmarking considered difficult: Common pitfalls in database performance testing," in *Proceedings of the Workshop on Testing Database Systems (DBTest)*. Houston, TX, USA: ACM, 2018, pp. 1–6.
- [27] P. K. Erdelt and J. Jestel, "DBMS-Benchmark: Benchmark and evaluate DBMS in Python," *Journal of Open Source Software*, vol. 7, no. 79, p. 4628, Nov. 2022.
- [28] T. Kalibera and R. Jones, "Rigorous benchmarking in reasonable time," in *Proceedings of the 2013 International Symposium on Memory Management (ISMM)*. ACM, 2013, pp. 63–74.
- [29] A. Vogelsgesang, M. Haubenschild, J. Finis, A. Kemper, V. Leis, T. Mühlbauer, T. Neumann, and M. Then, "Get real: How benchmarks fail to represent the real world," in *Proceedings of the Workshop on Testing Database Systems (DBTest)*. ACM, 2018, pp. 1–6.
- [30] A. van Renen, D. Horn, P. Pfeil, K. Vaidya, W. Dong, M. Narayanaswamy, Z. Liu, G. Saxena, A. Kipf, and T. Kraska, "Why TPC is not enough: An analysis of the Amazon Redshift fleet," *Proceedings of the VLDB Endowment*, vol. 17, no. 11, pp. 3694–3706, 2024.
- [31] Transaction Processing Performance Council, "Fair use of TPC benchmarks," https://www.tpc.org/TPC_Documents_Current_Versions/pdf/Fair_Use_Quick_Reference_v1.0.0.pdf, 2019, quick reference v1.0.0. Accessed 2026-08-31.
- [32] M. Raasveldt and H. Mühleisen, "DuckDB: An embeddable analytical database," in *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. ACM, 2019, pp. 1981–1984.
- [33] E. J. Williams, "Experimental designs balanced for the estimation of residual effects of treatments," *Australian Journal of Scientific Research Series A: Physical Sciences*, vol. 2, no. 2, pp. 149–168, 1949.
- [34] C. A. Field and A. H. Welsh, "Bootstrapping clustered data," *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, vol. 69, no. 3, pp. 369–390, 2007.
- [35] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.
- [36] PostgreSQL Global Development Group, "PostgreSQL 18 documentation: PREPARE," <https://www.postgresql.org/docs/18/sql-prepare.html>, 2026, accessed 2026-09-09.
- [37] L. Doshi, V. Zhuang, G. Jain, R. Marcus, H. Huang, D. Altinbüken, E. Brevdo, and C. Fraser, "Kepler: Robust learning for faster parametric query optimization," *Proceedings of the ACM on Management of Data*, vol. 1, no. 1, pp. 109:1–109:25, 2023.

APPENDIX A

COVERAGE AND EXECUTION RECORDS

A. Query Coverage and Censored Validation

Supplementary Table S4 lists each query's primary cell count. Tables S5 and S6 report exclusion of Commercial System A and large-overhead counts by index configuration for every dataset. Supplementary Note S1 identifies the censored cases and distinguishes direct from indexed-twin result evidence.

B. Measurement Provenance

Supplementary Note S1 records historical validation, plan-capture provenance, and superseded runs. Transaction inventories did not directly query the server version; unchanged package logs provide corroboration, not a per-call observation. The primary estimates use accepted timed outputs, excluding historical checks and instrumented plan timings.