ReSequel: Robust LLM-assisted Query Rewriting and Optimization using Templatization and Sampling

Saeed Fathollahzadeh
Concordia University
saeed.fathollahzadeh@concordia.ca

Essam Mansour
Concordia University
essam.mansour@concordia.ca

Matthias Boehm
TU Berlin & BIFOLD
matthias.boehm@tu-berlin.de

ABSTRACT

Heuristic query rewriting has long complemented cost-based optimization to improve performance. Such rewrites transform SQL queries into semantically equivalent forms that are easier or faster to execute. Examples are standardizing expressions, eliminating redundancy, propagating constants, pushing down selections and projections, unnesting queries, and utilizing constraints. Modern DBMSs implement hundreds to thousands of such rules, but maintaining them is notoriously difficult. The interactions among rules are complex, and their static nature and application order prevent adaptation to specific query and database characteristics. Recent approaches that use large language models (LLMs) for query rewriting show promise but face challenges regarding the large search space, reliable query verification, and exploitation of metadata. We present **ReSequel**, an *outer optimization layer* on top of existing DBMSs to rewrite SQL queries using LLMs. ReSequel leverages catalog and statistical metadata to infer *template-specific rules* that guide the LLM toward effective query transformations. We generate, verify, and rank rewritten query variants on sampled data to ensure result correctness and runtime improvements. Our experiments cover eight benchmarks: JOB, TPC-H, Stats(-CEB), Public BI, IMDB, DSB, and StackOverflow; multiple DBMSs: PostgreSQL, MySQL, and DuckDB; as well as LLM-based query rewriting baselines. ReSequel yields workload-level speedups of up to 16x over native DBMSs and 22x over LLM-based systems, with individual queries exceeding 600x, across eight benchmarks and three DBMSs.

PVLDB Reference Format:
Saeed Fathollahzadeh, Essam Mansour, and Matthias Boehm. ReSequel: Robust LLM-assisted Query Rewriting and Optimization. PVLDB, 19(10): 2880 - 2893, 2026.
doi:10.14778/3828612.3828639

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/CoDS-GCS/ReSequel.

1 INTRODUCTION

QUEL (in Ingres [43], based on tuple calculus) and SEQUEL [15, 16] (in System R [4], based on relational algebra) were competing early relational query languages. Ultimately, SEQUEL first became the de-facto standard and later evolved to the SQL standard. The success of SQL and the relational model—despite criticism on specific aspects [10, 71]—was largely due to its (1) declarativity (what not how), (2) flexibility (compose arbitrary complex queries), (3) freedom for automatic optimization, and (4) physical data independence. Despite these compelling properties, all DBMS remain stubbornly sensitive to suboptimal query formulations because query optimization is "a never-ending quest for an increasingly better model and repertoire of optimization and execution techniques" [93], various shortcomings and simplifying assumptions (e.g., the independence assumption and its implication for redundant, correlated predicates) [63], and brittle rewrites (e.g., for unnesting sub-queries) [70].

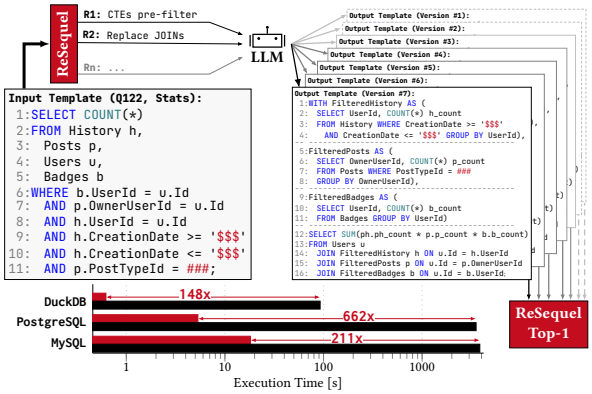


Figure 1: Comparison of ReSequel's rewriting impact on host DBMS performance. ReSequel rewrites masked templates into multiple variants using LLMs, and then locally verifies them, and selects the *Top*-1 reconstructed query.

Rule-based Query Rewriting: Many DBMSs have dedicated rule-based query rewriting systems, which transform queries into equivalent, but more efficient forms. These rule engines date back to the early 1990s [75, 76], contain 1,000s of rules, and are applied heuristically before the cost-based optimization of join orders and physical operators. Example rules are the elimination of `DISTINCT` if the result contains a `UNIQUE` attribute, query unnesting, and the propagation of constants over joins (for prefiltering both join sides):

$$R \bowtie_{R.a=S.b} (\sigma_{S.b>7}(S)) \rightarrow (\sigma_{R.a>7}(R)) \bowtie_{R.a=S.b} (\sigma_{S.b>7}(S)) . \quad (1)$$

Such rules are manually crafted by experts over decades [9] or automatically discovered by specialized tools [5, 92]. Since rules are expressed as source-target equivalences of small patterns, whole-query rewriting requires applying rules until a fixpoint or number of iterations. The interaction of rules and their application order renders development and debugging difficult and brittle, requiring considerable time, effort, and expertise. Also, the reliance on predefined rules makes rule engines inherently incomplete, potentially missing valuable rewrite opportunities for new queries [24].

Proceedings of the VLDB Endowment, Vol. 19, No. 10 ISSN 2150-8097.
doi:10.14778/3828612.3828639

LLM-based Query Rewriting: With the emergence of large language models (LLMs), new approaches to query rewriting appeared, which, however, still face fundamental limitations:

(1) *LLM-only:* First, there is work on LLM-based query rewriting that submits queries, optional metadata, and instructions to rewrite queries as a whole [3, 60, 86, 87, 100]. Due to the nature of LLMs, these approaches struggle with hallucinations, and the rewritten queries may underperform due to the lack of database characteristics.
(2) *LLM-based Rule Engines:* LLM-R2 [58] integrates Apache Calcite [9] rules, and employs LLMs to refine and select rules for whole query rewriting. This hybrid approach partially addresses the brittleness of rule systems, but the predefined rule set cannot exploit unforeseen opportunities.

These categories rely on unverified LLM reasoning or static rule sets, lacking reliability, robust improvements, and flexibility.

ReSequel Overview: We introduce ReSequel, a metadata-guided, LLM-assisted query rewriting system that operates as an *outer optimization layer* on top of existing DBMSs. We first gather rich data statistics and metadata on schemas, indexes, constraints, as well as supported DBMS features. ReSequel then generalizes queries into reusable *templates* with masked predicates (for scalable and cost-effective rewriting) and infers operation- and metadata-specific rules. These rules steer the LLM toward promising optimizations relevant for each template such as subquery unnesting, join reordering, or index creation. Each template and its rewriting tasks are encoded into structured prompts that instruct the LLM to generate alternative query variants. ReSequel verifies these variants on sampled data to ensure correctness and ranks them by performance, caching validated rewritten templates. This combination of template generalization, metadata-driven exploration, and sample-based verification enables ReSequel to produce high-performance queries in a robust and scalable manner. Figure 1 shows an example rewritten template and its performance impact on multiple DBMSs.

Contributions: Our primary contribution is the design of ReSequel, a practical framework for holistic query rewriting on top of DBMSs. Our main technical contributions are:

- *End-to-end Rewriting Workflow:* We present ReSequel's full pipeline, which integrates catalog metadata, DBMS statistics, and LLM reasoning for query rewriting, as well as verification and ranking of query variants (Section 3).
- *Template-based Query Generalization:* We introduce query *templates* that mask predicates and cluster similar queries for scalable rewriting. Templates preserve the query structures but enable reuse across queries (Section 4).
- *Sample-based Verification:* We employ database downsampling and query verification to validate semantic equivalence and select the top-performing query (Section 5).
- *Template Rewriting:* We describe the LLM-based template rewriting in terms of constructed prompts, query reconstruction, and query caching. (Section 6).
- *Experiments:* We study ReSequel's across eight benchmarks and three DBMSs, where we see speedups of up to 3–22× over DBMSs and 22× over LLM baselines (Section 7).

2 BACKGROUND AND CHALLENGES

We review traditional rule-based query rewriting systems and the challenges posed by messy real-world queries. We then formalize the problem of robust query rewriting for LLM-assisted systems.

2.1 Rule-based Query Rewriting

Traditional rule-based query rewriting frameworks [75, 76] consist of large collections of equivalence rules (source-to-target patterns) and a rule engine that transforms input queries into simpler or faster forms. Rewriting is typically applied after SQL parsing and semantic analysis but before cost-based query optimization. Rewrites are often implemented as hand-crafted C++ rules, enabling complex transformations such as subquery unnesting [76]. Despite their expressiveness, missed optimization opportunities are common. First, the ordering in which rewrite rules are applied is determined heuristically by experts. As rule sets evolve (e.g., to address specific bugs or performance issues) interactions between rules become difficult to reason about. An incorrect ordering may cause a rewrite to destroy the source pattern required by a more impactful rewrite. Second, rule sets are applied iteratively until a fixpoint or rewrite budget [76] is reached. Under restrictive budgets, large static rule sets often fail to trigger beneficial rewrites. This issue arises because many rules are irrelevant to a given query. Finally, many rewrite rules operate on few operators and are brittle with respect to source-pattern matching [63, 70]. Minor variations in query formulation, missing constraints or keys, or differences in operator placement can prevent rewrites from firing [62]. As a result, messy real-world queries remain challenging for rule-based systems.

2.2 Messy Query Characteristics

In contrast to many benchmarks, queries in the real world are often messy, originating from SQL-generating applications, text-to-SQL tools, or inexperienced users. Additionally, database schemas evolve (e.g., splitting structured attributes), yet queries remain unchanged.

Classification: We characterize common rewrite opportunities in messy queries using a hierarchical classification, shown in Figure 2. We distinguish Non-DBMS systems (external rewriting and learned optimizers) and DBMSs by labeling their support as fully supported, unsupported, or conditionally supported (+\-). Existing solutions address only specific cases, and support remains limited. We also report summary statistics of multiple workloads.

Non-DBMS Challenges: External optimizers primarily improve queries by applying predefined patterns and exploiting metadata, rather than discovering new query formulations. Accordingly, the rewrite effectiveness is highly sensitive to statistics, indexes, and constraints. Despite recent progress, several key challenges remain:

- *Scalability:* Most rewrite systems operate on individual queries, which does not scale to large workloads. E.g., LearnedRewrite [99] requires 16 hours to rewrite the IMDB workload, while workload execution takes only 11 hours. Scalability is even more challenging for LLM-based systems.
- *Cost:* The monetary cost of LLM-based rewrite systems is often prohibitive, caused by the large number of input/output tokens consumed during query rewriting.
- *Verification:* Rewrites may modify queries without guarantees of correctness, but only semantically equivalent queries

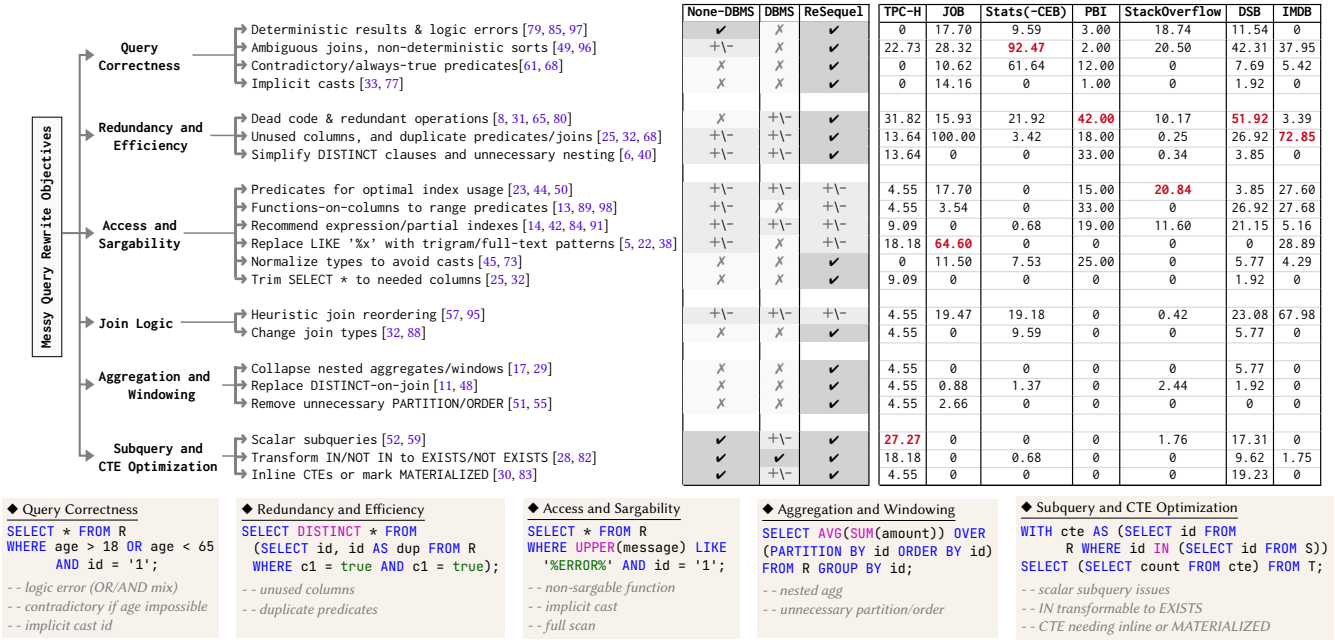


Figure 2: Challenges in Phase Ordering and Query Optimization Support in DBMS and Non-DBMS Systems.

should be accepted. Moreover, LLM-based systems often apply multiple transformations; for example, by introducing new functions, or extracting subqueries. Existing query verification systems neither support arbitrary transformations nor generalize well across SQL dialects.

- *Reusability and Robustness:* Most rewrite systems incur substantial upfront overhead, yet their outputs are not reusable, even for highly similar queries. Workload queries are typically instances of few templates, and thus, reusable.

DBMS Challenges: DBMSs primarily optimize via cost-based join ordering and operator selection as well as heuristic rewrite systems. However, complex or semantic rewrites often remain the user's responsibility because whole-query rewriting and cost-based search can be expensive. Rewrites may conflict with physical design. For example, applying a function to an indexed column (e.g., `EXTRACT(YEAR FROM date_col)`) can prevent index usage unless a matching functional/index expression exists.

2.3 Problem Formulation

To overcome DBMS and Non-DBMS limitations, LLMs offer a promising approach for rewriting messy, text-based queries in a holistic manner. For scalability, recurring query templates are optimized once and reused across query instances. This design enables thorough verification and ranking of query variants on samples.

Notation: Given an input SQL query $\mathbf{Q}$, our goal is to generate a semantically equivalent, but faster query $\mathbf{Q}'$. To this end, we define a query rewriting operator $R = [\mathcal{R}, \mathcal{F}, \mathcal{E}]$, where $\mathcal{R}$ is a set of predefined semantics-preserving rewrite rules, $\mathcal{F}$ is a set of interface functions which can be overwritten, and $\mathcal{E}$ are exploratory rewrite examples that enable discovering new transformations. The rewriting process is guided by an LLM, which applies R to the input query: $\mathbf{Q}' = R(\mathbf{Q})$. All rewritten queries must be semantically equivalent to the original query. To evaluate performance, candidate queries are executed on a downsampled database $\mathcal{S}$, and the final output query is selected by minimizing execution latency:

$$\mathbf{Q}' = \arg\min_{\mathbf{Q}_r \equiv Q} \ell(\mathbf{Q}_r, \mathcal{S}), \quad \text{s.t.,}\ \ell(\mathbf{Q}_r, \mathcal{S}) \leq \ell(\mathbf{Q}, \mathcal{S}).$$

3 RESEQUEL SYSTEM OVERVIEW

ReSequel is an end-to-end SQL query rewriting system designed as an outer optimization layer for existing DBMSs. It employs zero-shot, in-context learning to guide LLMs in generating semantically equivalent queries with improved performance. ReSequel relies on explicit task specification as well as operation- and metadata-aware guidance derived from system metadata, data characteristics, and DBMS-specific features. As illustrated in Figure 3, ReSequel consists of modular components that support query templatization, metadata management, guided rewriting, and scalable construction and evaluation of executable query variants. Rather than optimizing individual ad-hoc queries, ReSequel performs rewriting at the level of workload templates. To support this workflow, the system distinguishes between two layers: *online query processing* and *offline query rewriting*. Queries whose templates have been processed are handled by the online layer, which applies cached rewrites with minimal overhead. Queries of unseen templates are routed to the offline layer, where ReSequel performs metadata-driven rewrite generation, verification, and template materialization, enabling subsequent queries to benefit from reusable, optimized templates.

3.1 Query, Metadata, and Rewrite Hints

For an input query, we first collect metadata related to its parameters and associate it with categories of messy query characteristics. We extract the query template and model each query as a query tree composed of specific operators (e.g., Join, Filter, Projection). Based on the referenced tables, we prepare the metadata, including attribute characteristics, indexes, and primary and foreign keys. Finally, we check a set of predefined conditions to identify and label the query with specific characteristics, which can be used as rewriting hints. Figure 4 shows an example of these components.

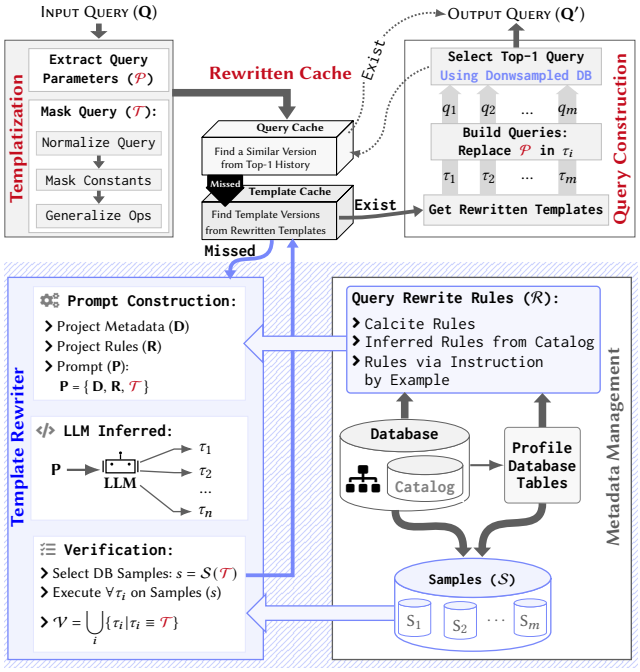


Figure 3: ReSequel Architecture and Workflow.

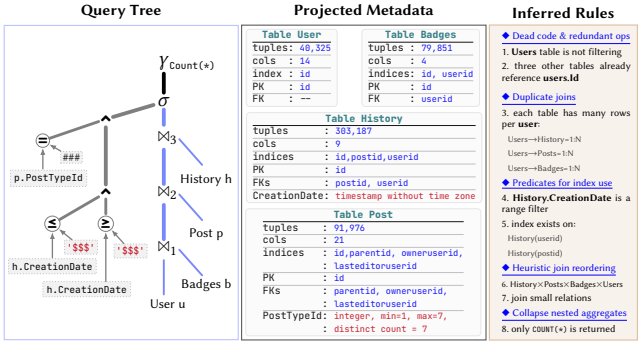


Figure 4: Materialization Steps for STATS Q#122 (Figure 1).

3.2 Online Query Processing

In the online layer, each incoming SQL query is templatized and then matched against cached templates to identify reusable optimized variants. Upon a cache hit, ReSequel reconstructs the query and returns an optimized version $\mathbf{Q}'$. Key online components are:

Templatization: Query workloads in production systems are rarely ad-hoc. They typically arise from a small number of parameterized SQL templates instantiated with different constants. For example, the IMDB workload contains 13,646 queries but only 79 distinct templates. ReSequel therefore rewrites templates instead of individual queries to achieve scalability and low overhead. It canonicalizes each incoming query by masking literal constants and normalizing syntactic variants. Queries with identical logical structure are mapped to a common template identifier that indexes cached templates. Templatization also extracts query parameters and aligns them with placeholders, enabling later reconstruction of query instances. ReSequel is designed to operate under high database load. Template-based rewriting is applied upfront while remaining flexible enough to handle previously unseen queries.

Caching: ReSequel maintains two caches to support efficient reuse. The `Query Cache` stores rewritten instances of previously optimized queries; upon a cache hit, ReSequel substitutes query parameters and immediately returns the optimized query $\mathbf{Q}'$. If no match is found, the query is matched against the `Template Cache`, which stores verified rewritten variants per template. On a template hit, ReSequel invokes query construction. Queries without cached templates are forwarded to the offline rewriting layer. Reusing optimized templates across workloads as well as the discovery of new built-in rewrites is interesting future work.

Query Construction: Given a set of rewritten and verified template variants ($\mathbf{t} = \{\tau_1, \tau_2, \ldots, \tau_m\}$), ReSequel reconstructs query instances by substituting the extracted parameters of the original query $\mathbf{Q}$ into the corresponding placeholders of each template variant. These candidate query instances ($\mathbf{q} = \{q_1, q_2, \ldots, q_m\}$) are subsequently evaluated and ranked on downsampled data, and we select the top-1 candidate as the rewritten query.

3.3 Offline Query Rewriting

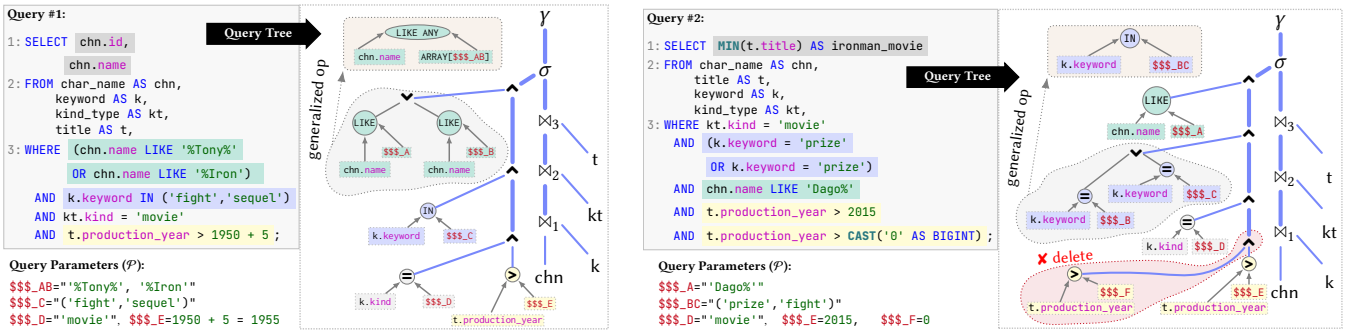
Previously unseen query templates undergo offline rewriting and verification. This process can be executed once upfront for a workload, asynchronously off the critical path, or on demand.

Metadata Management: The metadata management component creates a unified representation of metadata, rewrite rules, and verification samples required for template rewriting. This information is projected into LLM prompts to guide rewrite generation.

- *DBMS Catalog and Data Profiling.* ReSequel leverages DBMS catalog metadata, including schemas, constraints, and statistics, together with additional table-level profiling to support rewrite decisions that depend on data characteristics.
- *Query Rewrite Rules (R).* ReSequel incorporates static (heuristically applied) and cost-based (dependent on statistics) rewrite rules. The system integrates rules from query optimizers (e.g., Calcite [1]), profiled data, and curated, example-based guidance into a unified rule representation. We have collected nearly 50 examples of meta rules, crafted by hand to guide the LLM in query rewriting and exploratory search.
- *Database Downsampling (S).* To enable fast verification of rewritten queries, ReSequel constructs representative downsampled table clusters for each template.

Template Rewriter: ReSequel performs template rewriting to amortize rewrite costs across many query instances. It does not assume that a single rewrite is optimal for all template instantiations. Instead, the system selects the best variant per query instance based on actual predicate values during Top-1 selection. Given a template $\mathcal{T}$ and its associated metadata, ReSequel formulates a rewrite task and invokes the LLM. The prompt ($\mathbf{P} = \{\mathbf{D}, \mathbf{R}, \mathcal{T}\}$) combines projected metadata, rewrite rules, and the template text. The LLM generates template variants, which are verified on downsampled data. Valid variants are then stored in the template cache for reuse.

LLM Dependency: ReSequel collects metadata and rules, and generalizes workload queries into templates to reduce the reliance on LLMs. We do not directly trust LLM outputs. Instead, the LLM is used to create rewritten variants, but deterministic components validate them. Therefore, model evolution mainly affects the diversity of rewritten variants, not their ranking or correctness.



(a) Q1: A Query with a Generalized Operator

(b) Q2: A Query with a Generalized Operator and Redundant Filter

Figure 5: Example of Normalization, Masking, and Generalization of Operators for Two Queries / Parameter Materialization.

4 WORKLOAD TEMPLATIZATION

ReSequel's query rewriting is based on the templatization of queries for scalable and cost-effective rewriting. Figure 5 shows two example queries that are mapped to a single template. Templatization consists of two steps: template extraction and template refinement. ReSequel's templatization preserves the structural semantics of queries, including join graphs, grouping and aggregation operators, and predicate structure (columns and operators). Only literal constants and variable-length predicate lists are generalized. As a result, templatization defines the scope for rewriting.

Template Extraction: First, we extract the predicate values of individual filtered attributes and stored them in $\mathcal{P}$. Additionally, we refine the values by removing duplicate filter values, eliminating unnecessary conditions (e.g., `CAST('0' AS BIGINT) = 0`), and performing constant folding (i.e., executing and replacing simple arithmetic operations). This refinement is necessary because most database systems struggle with overly complex predicates. Subsequently, we generalize and mask the templates to create an abstract view $\mathcal{T}$ and thus, reduce the number of distinct templates. Algorithm 1 shows the pseudo-code for the two templatization steps. We first extract all predicate values from the queries in Line 3. ReSequel identifies a key pattern for each actual value (Line 6). This key pattern is a unique string obtained from the SQL query to extract the actual value or to find and replace it during query reconstruction. After finding unique keys for all values, we identify the operation type (e.g., `IN`, `LIKE`) between the value and referenced attribute (see Line 7), and materialize these key patterns in Line 8. ReSequel refines workload templates to produce a compact set of distinct templates. Structural constructs—such as outer joins with NULL-sensitive semantics, correlated subqueries, and window functions—are retained unmodified, allowing the LLM to apply rewriting rules over the full context. Order-sensitive and -insensitive queries are merged into a single template during templatization and separated during query reconstruction and Top-1 selection. We are using SQLGlot [64] to parse SQL queries into an AST and traverse it to identify operators to merge and extract predicate values. Therefore, dialect support, as well as support for specific constructs, depends on the capabilities provided by SQLGlot. A limitation lies in dialect-specific operator coverage: unrecognized constructs cannot be generalized into semantically equivalent templates. Broad and extensible support of SQL dialects is an interesting direction for future work.

Template Refinement: Second, ReSequel extracts query templates through a three-step refinement process. In Step 2a (Line 10),

Algorithm 1 TEMPLATIZATION($\mathbf{Q}$)

Input: SQL Query $\mathbf{Q}$
Output: Query Template $\mathcal{T}$, Query Parameters $\mathcal{P}$

1: $\mathcal{P} \leftarrow$ dict(); $\mathcal{T} \leftarrow \varnothing$;
2: *// Phase 1: Template Extraction*
3: V $\leftarrow$ EXTRACTQUERYVALUES($\mathbf{Q}$) *// get query parameter values.*
4: V′ $\leftarrow$ REFINEQUERYVALUES(V) *// e.g., remove duplicate values.*
5: **for** $v \in$ V′ **do** *// iterate over query values.*
6: key $\leftarrow$ GETKEYPATTERN($\mathbf{Q}, v$) *// key embedding.*
7: op $\leftarrow$ GETOPERATION($\mathbf{Q}$, key) *// operation identification.*
8: $\mathcal{P}$[key] $\leftarrow \{op, v\}$ *// add parameter value to $\mathcal{P}$.*
9: *// Phase 2: Template Refinement*
10: t $\leftarrow$ NORMALIZEQUERY($\mathbf{Q}$) *// query normalization.*
11: t $\leftarrow$ MASKQUERYVALUES(t) *// e.g., mask all strings to '$$$'.*
12: $\mathcal{T} \leftarrow$ GENERALIZEQUERYOPERATIONS(t, $\mathcal{P}$)
13: **return** $\mathcal{T}, \mathcal{P}$

we normalize the original queries by sorting the query outputs (since the order of projected columns is irrelevant), reordering join operations using a fixed alphabetical pattern for uniformity, and replacing table aliases with their original, non-conflicting names. To avoid complex variable handling, we extract a query tree from each query and derive the template from this tree. In step 2b in Line 11, ReSequel replaces all actual values with masked values. For example, we use the pattern `$$$_A` for strings and `###_A` for numerical values. After this step, we obtain a nearly static query template. Additionally, in step 2c in Line 12, we generalize the template's operations. Applications may inject flexible filter values (e.g., `LIKE ANY (ARRAY ['val1', 'val2', ...])`. If we keep the original operations, each different size of value lists would result in a distinct template. To generalize such lists, we replace them with a single masked value (e.g., `LIKE ANY (ARRAY [$$$])`). Similarly, for fixed operations such as `col1 = 1 OR col1 = 2 OR col1 = 3`, we generalize them into a list operation of $\mathcal{P}$ values and replace them with a single masked value. This refinement substantially reduces the number of query templates to rewrite. Templatization does not reason about rewriting query instance; it only ensures scalable and reusable rewriting, while later, we select optimal variants.

EXAMPLE 1 (WORKLOAD TEMPLATIZATION). *Figure 6 illustrates an example of extracting a template for the two SQL queries (in PostgreSQL dialect) from Figure 5. Here, distinct parts are highlighted in different colors. First, the queries contain different projection clauses,*

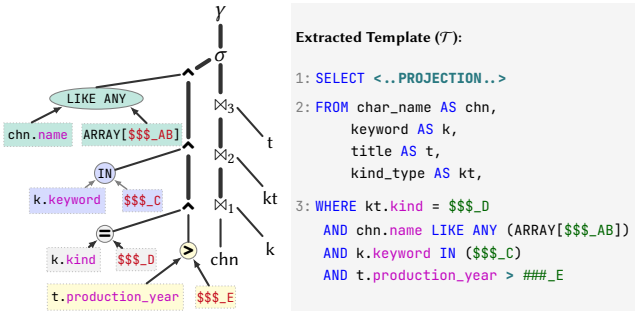


Figure 6: Extracted Template from Queries #1 & #2 in Figure 5.

which are masked during template extraction to enable structural clustering. The original projection (including projected columns and aggregation) is restored during query reconstruction and top-1 selection. Second, the order of clauses does not matter; for example, `kt.kind` *appears in different positions across queries. Third, ReSequel generalizes the* `LIKE`, `OR`, *and* `IN` *operators. At this stage, we do not attempt to determine whether* `k.keyword IN ('fight','sequel')` *performs worse than* `k.keyword = 'fight' OR k.keyword = 'sequel'`. *Our goal is simply to generalize such operations. This generalization preserves predicate semantics while avoiding template explosion due to syntactic variations. Fourth, the extracted query parameters reveal that the filter condition* `t.production_year > CAST('0' AS BIGINT)` *in Query #2 is redundant and can be removed. Fifth, ReSequel's template extraction and refinement enables merging queries into a single template when the same column appears with different operations. In Query #1 and Query #2, after removing unnecessary conditions, ReSequel applies outer generalization to merge* `LIKE` *and* `LIKE ANY` *into a unified leaf condition. The resulting query tree is then converted into an SQL template, and key patterns and operations are materialized for later query reconstruction.*

5 DATABASE DOWNSAMPLING

With the workload templates in hand, workload verification and costing requires executing actual queries over samples. ReSequel extracts multiple datasets with different schema subsets and stores them separately. We use downsampled datasets for three reasons: (1) to perform syntax checks of LLM-generated queries, (2) to verify LLM-rewritten queries for result correctness, and (3) to evaluate the runtime of verified candidate queries, and select the top-1 candidate per original query. Drawing multiple samples per cluster of tables ensures robust verification and performance evaluation. Algorithm 2 shows the overall downsampling process.

Downsampling Scope and Guarantees: ReSequel uses downsampled databases to filter invalid rewritten queries and to evaluate performance among candidate queries. Downsampling does not provide a formal guarantee of semantic equivalence or exact runtime behavior on the full database. To mitigate this limitation, ReSequel verifies candidates across multiple samples per schema cluster, and includes the original query as a fallback. For analyzing the downsampling trustworthiness, we conduct dedicated tests. We execute the queries on sample datasets and collect the results. Next, we randomly remove some query results from selected samples and re-executed the queries to verify that the outputs of rewritten and original queries remain identical. Here, we preserve predicate matches to avoid empty query results equivalence checking.

Algorithm 2 DATABASESAMPLING($\mathcal{L}, \mathcal{D}, C, \tau$)

Input: List of Templates $\mathcal{L}[\mathcal{T}_1, ..., \mathcal{T}_n]$, Database $\mathcal{D}$, Data Catalog C, Database Instance Limits τ
Output: Set of Databases $\mathcal{S}$

```
1:  X ← dict()                                   // template info dictionary.
2:  // Phase 1: Schema Selection
3:  for t ∈ L do                                 // iterate over workload templates.
4:      tbls ← GETTABLES(t)                      // get all table names in template.
5:      cols ← GETCOLUMNNAMES(t)                 // get all col names in template.
6:      schema ← GETSCHEMA(D, C, tbls)           // get template schemas.
7:      X[t] ← (tbls, cols, schema)
8:  // Phase 2: Schema Clustering
9:  C ← SCHEMACLUSTERING(X)                      // cluster templates.
10: // Phase 3: Database Sampling
11: S ← dict()
12: for c ∈ C do                                 // iterate over template clustrs.
13:     tbls ← GETTABLES(c)                      // get table names in cluster.
14:     cols ← GETCOLUMNNAMES(c)                 // get col names in cluster.
15:     schema ← GETSCHEMA(D, C, tbls, cols)     // get cluster schemas.
16:     S[c] ← CREATEDATABASE(D, tbls, cols, schema, τ)   // add DBs.
17: return S
```

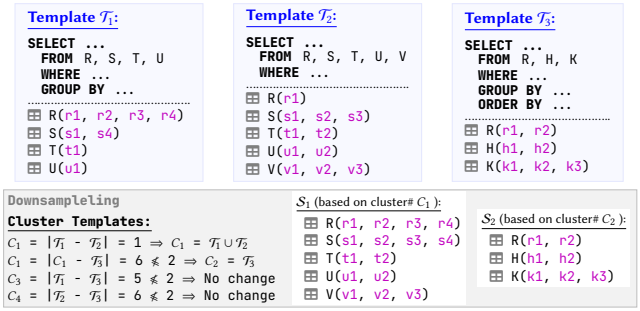


Figure 7: Example Template Clustering and Schemas.

Schema Selection: We first iterate over the list of templates ($\mathcal{L}$) extracted from the workload and auxiliary data catalog information (C). At the end, we obtain a subset of the schema relevant for the query workload. Specifically, in Line 3-5, we iterate over the templates, and extract all tables and columns accesses in these templates. Next, we prune the original database schema to the subset relevant to these accessed tables, columns, filter indexes, primary keys (PK), and foreign keys (FK) (see Lines 6 and 7).

Schema Clustering: To avoid creating a separate schema per template, we cluster templates by merging overlapping table sets (Line 9). We first cluster templates that access the same tables. Then, we add a template to an existing cluster if it differs by up to two tables. This clustering substantially reduces the number of schemas.

EXAMPLE 2 (TEMPLATE CLUSTERING AND SCHEMA CREATION). *Figure 7 shows an example of template clustering and related schema creation. ReSequel materializes the tables and columns necessary for projection, join, and filter operations. Templates $\mathcal{T}_1$ and $\mathcal{T}_2$ share four common tables but differ by one table. Consequently, we merged $\mathcal{T}_1$ and $\mathcal{T}_2$ into a single cluster, C_1. In the subsequent clustering iteration, we compare the updated cluster C_1 with template $\mathcal{T}_3$. Since the difference between C_1 and $\mathcal{T}_3$ exceeded our (configurable) threshold of two tables, we create a new cluster C_2 for $\mathcal{T}_3$. We also evaluate the pairs*

($\mathcal{T}_1$, $\mathcal{T}_3$) and ($\mathcal{T}_2$, $\mathcal{T}_3$), but because these pairs also exceed the threshold, and the tables were already covered by clusters C_1 and C_2, no changes are made to the clusters. For each cluster, ReSequel finally merges the columns of overlapping tables to create a unified set of columns. Additionally, we analyze the original data catalog and incorporate relevant physical design choices (e.g., indexes) into the sample schema.

Database Sampling: Downsampling and verification are performed per template and reused across query instances. Hence, their cost is amortized over the workload. After schema clustering, we generate a diverse set of databases per cluster. For instance, cluster C_1 may contain up to τ databases. This design serves the purpose of reducing risk of validating correctness and performance of queries on a single small sample. In Lines 12-16, we extract tables, columns, and the relevant schema for each cluster, then sample the database based on column dependencies. Line 16 (createDatabase) is the most costly step. We first identify the central table in the subset based on FK dependencies, that is the table most connected to others via outgoing or incoming FKs. We then take a uniform random sample of rows from this central table, and follow relationships to sample matching rows from connected tables to prevent the well-known issue of empty results over independent samples [34, 56]. Our sampling procedure also accounts for the following challenges:

- *Missing Relationships:* Ideally, subset tables preserve FK relationships, but some may be absent due to write-optimization or queries relying on non-key columns. In such cases, we analyze join predicates to infer relationships and leverage feature dependencies collected in the metadata.
- *Multiple Central Tables:* If multiple central table candidates exist (e.g., galaxy or snowstorm schemas), and since we generate multiple databases per cluster, we start from these candidates round robin for different samples.
- *Sample Selection:* We choose samples that cover diverse data characteristics, e.g., frequent and infrequent values as well as NULLs, to improve robustness across workload patterns.

6 QUERY REWRITING

With the query templates and sampled databases at hand, we are ready to rewrite queries. ReSequel is a prompt-based system that guides an LLM to produce optimized queries using rewrite rules, metadata, and task-specific instructions. Cleaning up syntactic issues (see Figure 2) typically requires one or two LLM iterations. However, optimizing queries is more challenging because runtime data is unavailable to the LLM. To address this issue, ReSequel constructs prompts in a structured manner: each prompt is derived from the query template and catalog metadata, and targets a single optimization task per iteration. Finally, all candidate rewritten queries are verified through execution, and the best candidate is selected based on its performance on sampled databases.

6.1 Overall Template Rewriting

Algorithm 3 outlines the end-to-end rewriting of a query workload. We assume a streaming model, where each incoming query is processed independently; static workloads are handled by iterating over all queries. To minimize redundant computation, ReSequel employs a two-level cache: a `Query Cache` for rewritten queries and a `Template Cache` for validated rewritten templates (Figure 3).

Algorithm 3 queryRewrite($\mathbf{Q}, \mathcal{M}, \mathcal{D}, \tau$)

Input: Raw Query $\mathbf{Q}$, LLM $\mathcal{M}$, Database $\mathcal{D}$, Instance Limits τ
Output: New Query $\mathbf{Q}'$

1: $\mathcal{T}, \mathcal{P} \leftarrow$ templatization(q) *// get query template and params.*
2: $\mathbf{Q}' \leftarrow$ reconstructQuery($\mathbf{Q}, \mathcal{T}, \mathcal{P}$) *// retrive from cache.*
3: **if** $\mathbf{Q}' = NULL$ **then** *// Rewritten Cache missed.*
4: $C \leftarrow$ readAndBuildCatalog($\mathcal{D}$) *// build data catalog.*
5: $\mathcal{S} \leftarrow$ databaseSampling($\mathcal{T}, \mathcal{D}, C, \tau$)
6: prompt $\leftarrow$ prompt($\mathcal{T}, C, \mathcal{M}$) *// get rewrite prompt.*
7: $\mathcal{R} \leftarrow$ submitPromptToLLM(prompt, $\mathcal{M}$) *// t={t_1, ..., t_n}.*
8: O $\leftarrow$ runQuery($\mathbf{Q}, \mathcal{S}$) *// execute orig query on sampled DB.*
9: **for** $r \in \mathcal{R}$ **do** *// iterate over raw rewritten templates.*
10: q $\leftarrow$ buildQuery($r, \mathcal{P}$) *// replace params on template.*
11: **if** runQuery(q, $\mathcal{S}$) = O **then** *// all results identical.*
12: TemplateCache[$\mathcal{T}$] $\leftarrow \bigcup$ r *// update Template Cache.*
13: $\mathbf{Q}' \leftarrow$ reconstructQuery($\mathbf{Q}, \mathcal{T}, \mathcal{P}$) *// get Top-1 query.*
14: **return** $\mathbf{Q}'$

Query Templatization: For an input query, we first extract its parameterized template and parameter bindings using Algorithm 1, Line 1. Subsequently, we attempt query reconstruction by probing the Query Cache to retrieve a previously rewritten query. A cache miss—typical for unseen templates or during warm-up—triggers the LLM-based rewriting pipeline (Lines 3-13).

Database Sampling As part of metadata management (Line 4), we construct an extended data catalog that augments the native DBMS catalog with column-level statistics and inferred dependencies not explicitly declared in the schema. Additionally, we execute Algorithm 2 to generate samples for template clusters.

Rewriting and Evaluation For each extracted template, we construct concise, task-specific prompts (Line 6) and submit them in batch to the LLM, producing multiple candidate rewritten templates (Line 7). The LLM is used exclusively to generate candidate rewritten templates; correctness and performance decisions are enforced by execution-based validation and selection. Specifically, we execute the original query and reconstructed queries for each template variant on the sampled database and compare results. Only candidates with identical results are admitted into the `Template Cache` (Line 12). Subsequent invocations benefit from cache hits, and we return the Top-1 candidate as final rewritten query (Line 13).

6.2 Prompt Construction

For guiding the LLM to efficiently generate effective versions of query templates, we aim at global template optimization, where generated versions should cover more than one query. The generated prompts include the masked template, tasks and instructions, example optimization rules, as well as metadata and statistics.

Tasks and Instructions: Since we do not know upfront which information is needed for the LLM's rewriting process, we base our approach on optimization tasks and instructions. We start from operations of a query template and create tasks for types of optimizations. Table 1 shows an example of the tasks and instructions, verified against the schema, and used to iteratively prompt the LLM based on the associated examples and metadata. ReSequel includes instruction prompts based on existing operations in the template

Table 1: The prompts used to generate the optimized template version are based on tasks and operations. In addition to the prompt, we also encode the schema (e.g., relevant tables, columns, and indexes) as well as statistics in the requests.

V#	Prompt Content (Template SQL Script + Projected Schema + Metadata + Examples)	LLM Recommend Example (Implemented Functions + New Template)
$V1$	Rewrite SQL query for optimization, possibly using **data skipping through prefiltering**.	`CREATE INDEX trgm_idx ON name USING gin (name gin_trgm_ops);`
$V2$	Rewrite SQL query for optimization by **recommending appropriate data structures**.	`SET LOCAL enable_nestloop = off; SET LOCAL enable_hashjoin = on;`
$V3$	**Implement functions** to optimize queries involving **equality (=) conditions**.	`CREATE INDEX idx_hash_role ON role_type USING HASH (role);`
$V4$	Rewrite the SQL query based on **join types and filter operations** ($p1$).	Push filters to scans, replace cross joins with explicit INNER JOINs and use EXISTS for many-to-many links.
$V5$	Optimize the query by **implementing a join cardinality estimation formula** ($p2$).	Formula: (rows_A * rows_B) / max(distinct_A, distinct_B).
$V6$	Enables **probabilistic data structures** for highly efficient Joins($p3$).	`CREATE EXTENSION IF NOT EXISTS bloom;`
$V7$	Rewrite the template to **optimize subsequent filtering**.	Optimized with a CTE for the filtered table and EXISTS clauses.
$V8$	**Candidate Key Identification**: Analyzes the provided schema statistics to identify columns where the count of distinct values.	`CREATE FUNCTION find_candidate_keys(table_name TEXT)`

Algorithm 4 RECONSTRUCTQUERY($\mathbf{Q}, \mathcal{T}, \mathcal{P}$)

Input: Query $\mathbf{Q}$, Query Template $\mathcal{T}$, Query Parameters $\mathcal{P}$
Output: New Query $\mathbf{Q}'$

1: $\mathbf{Q}' \leftarrow$ QUERYCACHE$[\mathcal{T}, \mathcal{P}]$ *// read Query Cache.*
2: $\mathcal{R} \leftarrow$ TEMPLATECACHE$[\mathcal{T}]$ *// read Template Cache.*
3: **if** $\mathbf{Q}' \neq NULL$ **then** *// query cache hit.*
4: **return** BUILDQUERY(t, $\mathcal{P}$) *// reconstruct query by cache.*
5: **else if** $\mathcal{R} \neq NULL$ **then** *// template cache hit.*
6: $\mathcal{S} \leftarrow$ READDATABASESAMPLING($\mathcal{T}$)
7: $\mathcal{L} \leftarrow [\mathbf{Q}]$
8: **for** $r \in \mathcal{R}$ **do** *// iterate over verified rewritten templates.*
9: q $\leftarrow$ BUILDQUERY($r, \mathcal{P}$) *// replace masks with params.*
10: $\mathcal{L} \leftarrow \bigcup$ q
11: $\mathbf{Q}' \leftarrow$ TOP-1($\mathcal{L}$) *// rank queries on sample DBs.*
12: QUERYCACHE$[\mathcal{T}, \mathcal{P}] \leftarrow \mathbf{Q}'$ *// update Query Cache.*
13: **return** $\mathbf{Q}'$
14: **else**
15: **return** $NULL$

and attaches metadata related to these instructions as user messages. Additionally, and in order to allow for exploring a diversity of rewritten queries, we provide examples of possible optimizations as rules. These examples are not limited to the datasets, and the LLM is asked to generate several versions of the template.

6.3 Query Reconstruction and Caching

Following LLM-based rewriting, ReSequel invokes Algorithm 4 to reconstruct concrete queries by instantiating predicate values and to evaluate their performance. ReSequel first probes the `Query Cache`. If a rewritten instance of the same template with similar parameter bindings exists, the system retrieves the cached rewritten query (Line 1), instantiates it with the current parameters (Line 4), and returns the query. Upon a cache miss, ReSequel retrieves the set of verified rewritten templates produced by Algorithm 3 (Line 2). We then construct a candidate set by instantiating each rewritten template with the original parameter values and adding the original query as a baseline candidate (Lines 7-10). These candidate queries are evaluated on the downsampled database (Line 6). Execution is performed in parallel with early termination: once a candidate completes, the remaining executions are aborted, preventing excessive overhead from poorly performing queries (Line 11). Although this evaluation is conducted on sampled data and may yield different execution plans, our sampling strategy empirically produces near-optimal selections. Finally, the fastest candidate is inserted into the `Query Cache` for reuse (Line 12) and returned as the rewritten query. If no verified rewritten templates are available, the procedure returns `NULL`, indicating that no rewrite can be applied.

6.4 System Limitations

ReSequel focuses on query rewriting for read-only workloads and assumes independent query execution. Its database downsampling strategy works well when queries access subsets of tables and columns. It becomes more challenging for queries that span the entire database, where preserving the data distributions is difficult. In addition, queries with side effects, such as user-defined functions that perform updates, are not explicitly handled.

7 EXPERIMENTS

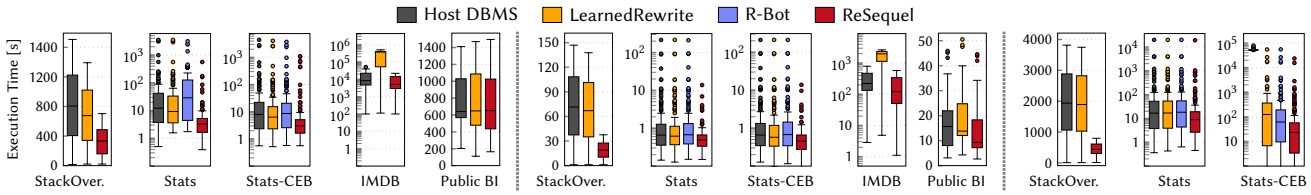
We evaluate ReSequel on a diverse set of benchmarks, including real-world datasets, across multiple DBMSs and LLMs.

7.1 Experimental Setup

Implementation Details: ReSequel's rewriting, verification, and top-1 selection is implemented in Python, whereas the downsampling is written in C++ (which reduced the overhead by up to 15x). For rewriting, we use Google AI Studio (Gemini-2.5-pro) [36] and Groq Cloud (OSS-120B) [2], whose latency can vary substantially (across LLMs and peak times). To build the data catalog, we extract the host DBMS's catalog, and execute SQL data profiling queries to obtain statistics. ReSequel currently supports three DBMS dialects: PostgreSQL, MySQL, and DuckDB.

HW/SW Environment: We ran all experiments on a VM with an Intel CPU (32 vCPUs) and 148 GB DDR4 RAM. The software stack comprises Ubuntu 22.04, Python 3.10, C++ 17, PostgreSQL 17.1, MySQL 8.0.43, and DuckDB 1.4.0. We tuned PostgreSQL using PGTune [72] for general OLAP workloads. For MySQL, we configured InnoDB with `buffer_pool_size=100GB`. DuckDB requires no special tuning. All DBMSs were run with multi-threading enabled.

Benchmarks: We evaluate ReSequel on the TPC-H (sf=10) [19], DSB (sf=100) [20], Stats/Stats-CEB [41], Public BI Benchmark [39] (top 100 longest-running queries), Join Order Benchmark (JOB) [54], IMDB Full [69], and StackOverflow [81]. Our objectives are to assess the performance benefits of LLM-assisted rewriting, and compare template- and query-based rewriting. We use three workload groups: *Group 1*, where templates and queries are identical except for masked parameters (TPC-H, Stats/Stats-CEB, DSB, Public BI); *Group 2*, where $\approx$ 15% of queries map to the same template (JOB); and *Group 3*, subsets of 1,192 queries from SQLStorm for StackOverflow, and 13,646 from IMDB, with high overlap and few templates. Table 2 summarizes the data and workload characteristics.



(a) PostgreSQL (b) DuckDB (c) MySQL

Figure 8: Overall Performance Impact on Various DBMSs Across 5 Datasets (LLM: Gemini-2.5-pro; 10% Downsampling).

Table 2: Used Datasets and their Workload Characteristics.

Dataset	# Tables	Workload	# Queries	# Templates
TPC-H	8	TPC-H (sf=10)	22	22
DSB	25	DSB (sf=100)	52	52
Public BI	83		100	100
Stats/Stats-CEB	8		146	146
JOB	21		113	95
StackOverflow	8	SQLStorm	1,192	60
IMDB	21		13,646	79

Table 3: Ratio of Rewritten Queries (Gemini & PostgreSQL).

Baeline	TPC-H	DSB	Public BI	Stats	Stats-CEB	JOB	StackOverflow	IMDB
LLM-R2	95%	61%	N/A	N/A	N/A	92%	N/A	N/A
R-Bot	95%	5%	0%	15%	15%	60%	N/A	N/A
LR	86%	63%	72%	12%	12%	74%	75%	37%
ReSequel	100%	100%	100%	100%	100%	100%	100%	100%

Table 4: Ratio & Verification Time of SQLSolver (PostgreSQL).

	TPC-H	DSB	Public BI	Stats	Stats-CEB	JOB	StackOverflow	IMDB
Equal [%]	27	0	0	8	7	0	0	0
Unknown [%]	73	100	100	92	93	100	100	100
Exe. Time [min]	72.5	1,142	5,764	2.1	1.7	0.6	3.7	59.6

Baselines: Furthermore, we consider three types of baselines: (1) a *Host DBMS systems*, where we run unmodified queries; (2) two *LLM-based systems (LLM-R2 [58] and R-Bot [86])* that use LLMs to apply rules; and (3) a *Non-LLM-based, learned-rewrite system (LearnedRewrite (**LR**) [99])*, which rewrites queries using a cost estimation model and Monte Carlo Tree Search to identify the best query. To verify the equivalence of ReSequel-rewritten queries, we used SQLSolver [21] as a baseline for ReSequel downsampling verification. R-Bot and LLM-R2 are methods that rewrite individual queries and therefore cannot handle workloads with many queries. In addition, LLM-R2 requires training on query plans, which is not available for all benchmarks. Therefore, we ran R-Bot for Groups 1 and 2 of workloads, and LLM-R2 for JOB, TPC-H, and DSB.

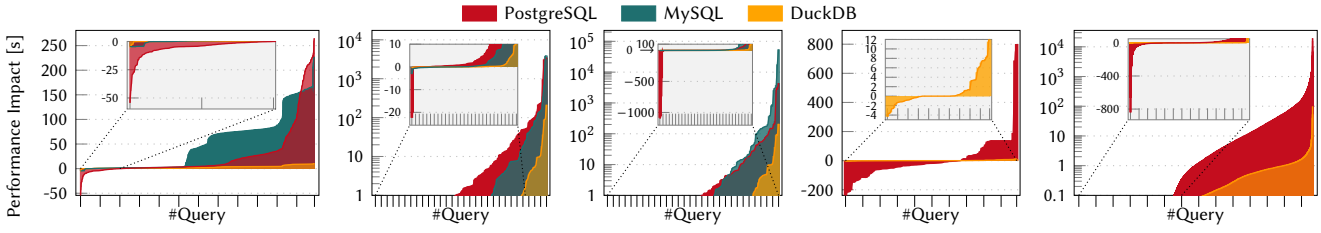
7.2 End-to-End Experiments

We evaluate both the overall and individual query performance.

Rewriting Robustness: To illustrate the effectiveness of the baselines in modifying workload queries, Table 3 reports the ratio of successfully modified queries. This ratio is measured on PostgreSQL; the results for MySQL and DuckDB differ by at most $\approx 1\%$. The LLM-based baselines and LearnedRewrite show limited success in rewriting workloads of queries while preserving query equivalence, and they also fail to support workloads with many queries. Here, we selected five datasets for which the baselines exhibited relatively high overlap. Additionally, Table 4 summarizes SQLSolver performance (outputs: Equal, Unknown). The results show most LLM-generated queries are unsupported by equivalence checkers like SQLSolver because they often fall outside the recognized AST patterns. Additionally, execution time can exceed four days for some workloads (e.g., Public BI), rendering it substantially slower than ReSequel's downsampling-based verification.

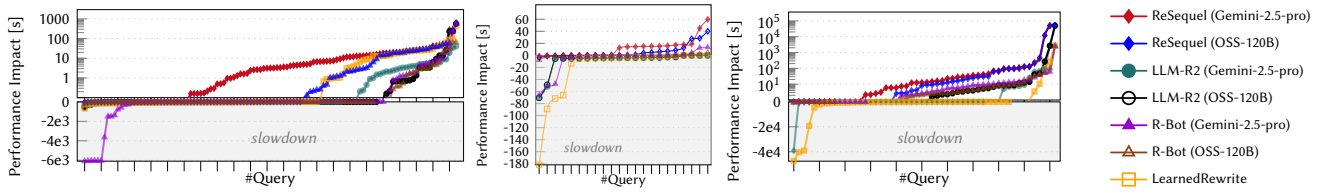
Workload Performance Impact: Figure 8 (datasets from Groups 1 & 3) presents the overall performance impact of ReSequel's rewriting. First, on StackOverflow and IMDB, ReSequel merges many queries into a small number of templates (StackOverflow: 60 templates for 1,190 queries; IMDB: 79 templates for 13,646 queries). The overall workload execution time improved by up to 2.45x on PostgreSQL, 4.81x on MySQL, and 3.95x on DuckDB. The StackOverflow workload—generated using GPT-4o-mini—contains many messy queries, where existing rewrite systems often struggle. Similarly, the IMDB workload includes redundant join orders, which makes cost-based optimization difficult. LearnedRewrite rewrites queries individually and achieves limited gains, as its Calcite rules also struggle with such messy queries. In contrast, ReSequel achieves substantial improvements through rewrites such as early string filtering, data-structure hints, dynamic join ordering, redundant join elimination, and subquery simplification and reuse. Second, for Stats, Stats-CEB, and Public BI, only literal values were masked. On Stats and Stats-CEB, ReSequel improved execution time by 5.89x/4x on PostgreSQL, 14.85x/16x on DuckDB, and 3.3x on MySQL (Stats-CEB). These gains stem from generating up to 20 alternative query variants per input query, breaking the original query into parts (some operating on data and others on the catalog), and removing redundancies in aggregation queries (e.g., redundant `GROUP BY` clauses). On Public BI, ReSequel shows limited improvement. The dataset is large (386 GB), which restricts downsampling diversity and weakens Top-1 selection. Many queries also use range predicates (where masking leads to over-generalized filters) and self-joins (where rewrite hints occasionally degrade performance). R-Bot successfully rewrites 12% of the Stats/Stats-CEB workload, but with negative performance impact. Although R-Bot is able to run on the Public BI workload, it incurs substantial overhead and fails to successfully rewrite even a single query (see Table 3).

Query Performance Impact: Figures 9 and 10 compare the performance of rewritten and original queries. Templatization consistently improves performance, with most queries experiencing speedups. These results show only the performance of rewritten queries, whereas the rewriting overheads are discussed in Section 7.3. Although a few queries experience slowdowns, their impact on overall performance is negligible. Several high-cost queries of StackOverflow and IMDB achieve substantial improvements. Public BI remains the only workload with limited gains. However, many queries on DuckDB still improve due to reduced numbers of intermediates. In contrast, LLM-based systems and LearnedRewrite improve only few queries and often incur slowdowns.



(a) StackOverflow (b) Stats (c) Stats-CEB (d) Public BI (e) IMDB

Figure 9: Individual Performance Impact on Workloads of Five Datasets (Overall = Figure 8, LLM = Gemini-2.5-pro).



(a) JOB (b) TPC-H (c) DSB

Figure 10: Individual Performance Impact on Workloads of Three Datasets (Overall = Figure 12, DBMS = PostgreSQL).

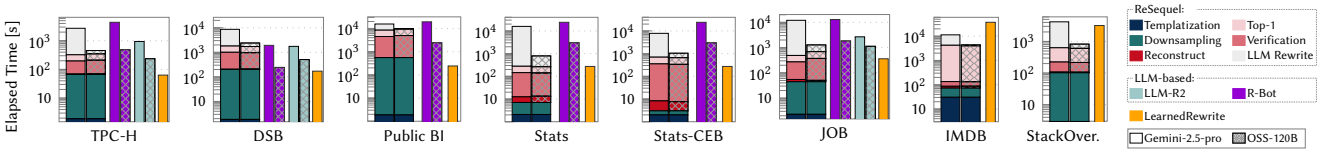
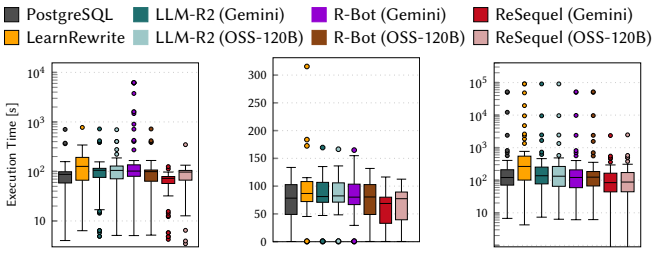


Figure 11: Time Breakdown of Rewriting Across Eight Datasets (Downsampling Rate = 10%, DBMS = PostgreSQL).



(a) JOB (b) TPC-H (c) DSB

Figure 12: Performance Impact on Three Datasets.

Impact of Different LLMs: Figure 12 shows the execution time of ReSequel on JOB, TPC-H, and DSB using two LLMs (Gemini and OSS-120B). On JOB, PostgreSQL already performs well for most queries, but several complex queries (e.g., 20a-20c) contain redundant joins (14 joins over 10 tables) and benefit from ReSequel's rewrites (Figure 12(a)). These improvements come from single-table subqueries combined with `INTERSECT`. LearnedRewrite yields small gains through join reordering, while LLM-R2 often produces suboptimal plans with very large intermediates (>200 GB). Most JOB queries rewritten with Gemini exhibit slowdowns. ReSequel achieves speedups of 6.18x, 5.64x, 5.76x, and 50x over LearnedRewrite, PostgreSQL, LLM-R2, and R-Bot. For TPC-H (Figure 12(b)), the workload diversity is low and thus, rewriting opportunities are limited. ReSequel still improves performance by 1.14x over PostgreSQL. LearnedRewrite and LLM-based systems show minimal or negative gains. Differences from previously reported results are due to the underlying LLMs (Gemini and OSS-120B instead of GPT-4), indicating strong sensitivity. The DSB workload (Figure 12(c)) is skewed and highly correlated. Here, LLM-R2 and R-Bot perform worse than with GPT-4. ReSequel remains robust by generating multiple candidate rewrites per query. This strategy increases the chances of at least one variant showing improvements. Overall, ReSequel achieves speedups of 38.6x, 22x, 38.56x, and 22x.

7.3 Micro Benchmarks

In order to further understand these end-to-end results and the source of improvements, we conduct a variety of micro-benchmarks.

Time Breakdown of Rewriting: Figure 11 shows the total and per-step rewriting time of ReSequel and baselines (in log scale) across all datasets on PostgreSQL using Gemini-2.5-pro and OSS-120B. ❶, workload templatization is lightweight and negligible. ❷, building the extended data catalog is the main bottleneck; for fairness, timing reflects the full dataset even though smaller versions would suffice. ❸, the LLM rewrites templates; generation time scales with template count, and Gemini is slower than OSS-120B due to its thinking mode. ❹, workload reconstruction replaces placeholders with original values; runtime depends on query complexity, and topological reconstruction handles duplicate keys. ❺, during verification, the original query and up to N reconstructed variants per template are executed on sampled databases; total time grows with template count and N. ❻, Top-1 selection chooses the fastest candidate per query with parallel execution, early stopping, and caching. Differences on other DBMSs stem mostly from catalog construction, verification, and top-1 selection.

Number of Cache Misses: ReSequel uses template and query caches to reduce the number of LLM calls and amortize the top-1 selection. For datasets in Groups 1 and 2, all queries and templates result in 100% cache misses. For StackOverflow and IMDB, which have a large number of queries, only a few templates are rewritten. Figure 15 shows the frequency of selecting rewritten template versions during workload execution. For example, in IMDB, for template IDs 3, 4, and 5, all queries select version #1.

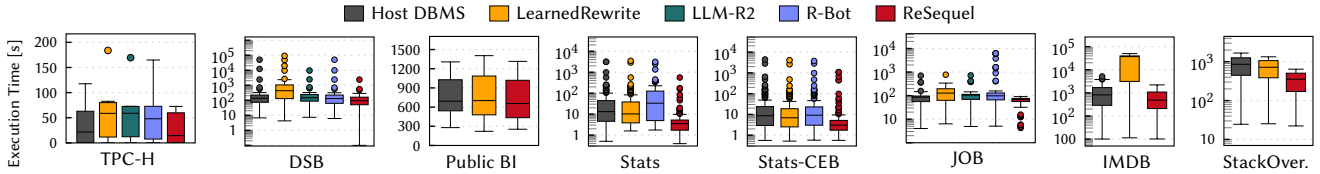
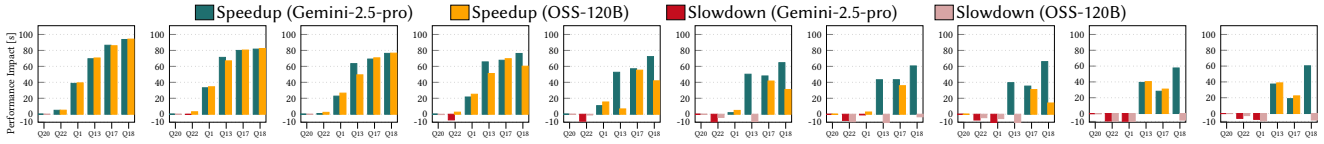


Figure 13: Impact of Queries with Messy Characteristics Across Datasets (Gemini-2.5-pro & PostgreSQL).



(a) Ratio=10% (b) Ratio=20% (c) Ratio=30% (d) Ratio=40% (e) Ratio=50% (f) Ratio=60% (g) Ratio=70% (h) Ratio=80% (i) Ratio=90% (j) Ratio=100%

Figure 14: Individual Query Performance Impact Debugging with Variety of Downsampling Ratios.

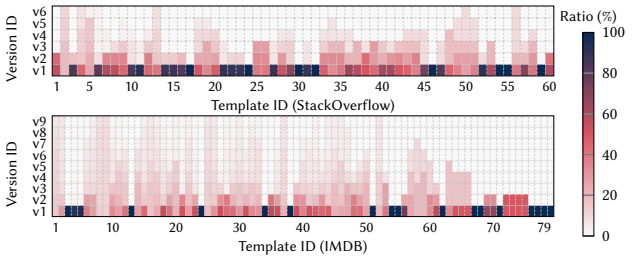
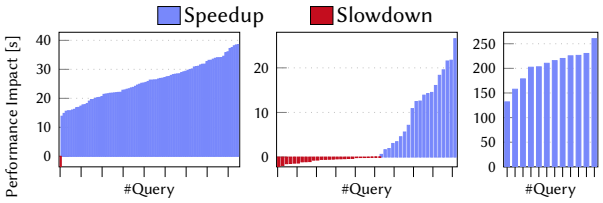


Figure 15: Number of Template and Query Cache Hits.



(a) |Queries|=86 (b) |Queries|=46 (c) |Queries|=12

Figure 16: Individual Query Performance Impact across Three StackOverflow Templates (DBMS: PostgreSQL; LLM: Gemini-2.5-pro; Downsampling Ratio = 10%).

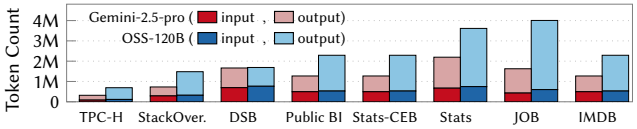


Figure 17: Costs in terms of Number of LLM Tokens for Rewriting all Query Templates of the Different Benchmarks.

Template-based Performance: We next evaluate the impact of template-based rewriting using Top-1 selection using three templates from StackOverflow. Figure 16(a) shows the template with the most merged queries, Figure 16(b) represents a mid-sized template, and Figure 16(c) the one with fewest merged queries. ReSequel performs well at both ends of this spectrum. Only one query shows a minor slowdown, while most merged queries improve. In contrast, for the mid-sized template, nearly half of the queries slow down due to aggregations. Here, Top-1 selection on a 10% sampled database fails to identify the optimal variant. Overall, these results indicate room for improvement. However, we observe no direct correlation between the degree of templatization (i.e., number of merged queries) and the performance of rewritten queries.

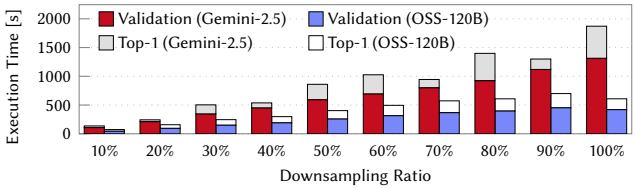


Figure 18: Validation and Top-1 Selection Overhead with Different Downsampling Ratios (TPC-H SF10; PostgreSQL).

Table 5: Count of Verified/Failed Rewritten Queries.

LLM	Q1	Q13	Q17	Q18	Q20	Q22	Total (Verified/Failed)
Gemini-2.5	8/12	18/2	17/3	20/0	10/10	14/6	120 (87+33)
OSS-120B	9/11	16/4	15/5	18/2	10/10	2/18	120 (70+50)

Rewriting Costs: LLM costs scale with the number of input and output tokens. Figure 17 reports the tokens for generating ten variants per template. To exceed an LLM's single-call output limit, we chain multiple requests, which increases token usage due to repeated instructions. For both LLMs, ReSequel uses identical prompts. Output tokens vary by model and include generated SQL queries, explanations, and hidden reasoning tokens. Across eight datasets, ReSequel consumes 6,142,842 tokens with Gemini and 12,085,412 tokens with OSS-120B. In contrast, baselines that invoke the LLM per query are less efficient. For example, rewriting 1,190 StackOverflow queries with Gemini requires over 14 million tokens.

Specific Messy Query Characteristics: In Figure 2, we highlight the fractions of messy query characteristics per dataset using red bars (e.g., Stats exhibits 92% ambiguous joins). Figure 13 shows the corresponding labeled characteristics. We select a subset of queries from each workload and execute them. Compared with the host DBMS and baselines, ReSequel effectively overcomes issues caused by messy queries and outperforms nearly all baselines.

Downsampling Ratios: To study the effect of sampling on validation and Top-1 selection, we use TPC-H SF10 and vary the downsampling ratio from 10% to 100%. We analyze six queries: {Q13, Q17, Q18} with long execution times and {Q1, Q20, Q22} with performance degradation at 10% sampling ratio. Figure 14 shows the performance impact, and Figure 18 shows the validation results. We evaluate two LLMs and observe that Gemini is substantially slower. However, as shown in Table 5, Gemini produces more successfully validated variants than OSS-120B. In contrast, most variants generated by OSS-120B show minimal or no performance impact, leading to faster validation and Top-1 selection. We always include

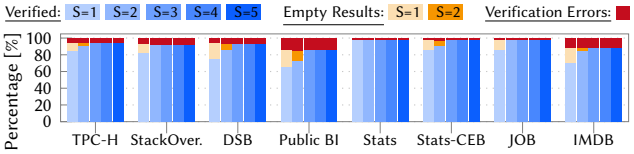


Figure 19: Verification Rates Across Different Sample Sizes (Verification Errors: incorrect LLM rewrites + reconstruction errors).

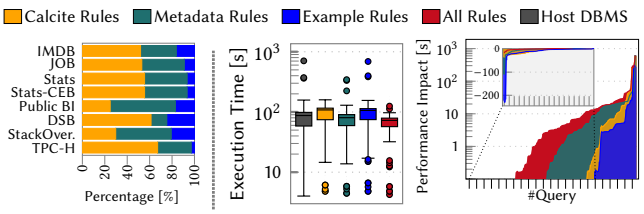
the original query among the candidates, which reduces the risk of performance regressions. OSS-120B achieves weaker runtime improvements due to fewer validated variants.

Impact of #Samples: To study the impact of the number of samples on query verification, for each workload, we generate 1-5 samples, execute the workload, and examine the queries. Figure 19 shows the ratio of verified queries, empty results (which are difficult to verify), and errors, i.e., unverified queries (w/ Gemini and PostgreSQL). The queries with errors were identified during the verification stage and simply deemed invalid variants. These results show that, in most cases, two (sometimes three) samples are sufficient to correctly verify the entire workloads.

Impact of Rewrite Categories: To study the sources of rewrite guidance in ReSequel, we performed an ablation study over the three rule categories: (1) *Calcite rules*, i.e., generic rewrite rules from Apache Calcite; (2) *metadata-derived rules*, i.e., rewrite guidance from schema information and workload metadata; and (3) *instruction-by-example rules*, i.e., manually curated rewrite examples. We keep the generation, verification, caching, and top-1 selection fixed, while varying only the rule guidance to the LLM. Figure 20(a) shows the distribution of rule categories applied by the LLM. Additionally, we decomposed every JOB query into three queries for the different rule categories. Figures 20(b) and 20(c) show that overall, ReSequel achieves a 5.64x speedup over PostgreSQL. With Calcite rules only, the speedup decreases to 4.4x, indicating that standard algebraic rewrites remain important. Metadata-derived rules only achieve a 2x overall speedup. These rules have the largest impact when transformations depend on schema semantics, relational properties, or workload context. However, in some cases, they negatively affect performance. Example-based rules primarily improve performance for long-running queries and queries requiring multiple transformations. In some cases, they introduce greater slowdowns. Overall, each category exhibits substantial performance variability, but applied together, these—sometimes overlapping—rules are a powerful augmentation for LLM guidance.

8 RELATED WORK

Query Rewrite Systems: Early work on query rewriting dates back to the 1990s. An example is the Starburst rewrite system [75, 76], which applies heuristic rules through a dedicated rule engine. Such systems support rewrites including common subexpression elimination, DISTINCT push-down and pull-up, predicate push-down, and subquery unnesting into joins. Subsequent work extended rule-based rewriting to new data models and queries, such as object-oriented queries, XML [35, 53], and text [7]. More recently, similar rule-based rewriting is used in ML systems [12] and DNN frameworks [27, 47]. In this setting, approaches such as super-optimization [46] and sum-product optimization [26, 90] automatically derive new rewrites from algebraic operator properties. In contrast to these general-purpose rewrite systems, we focus on



(a) Rules Ratio (b) Overall (JOB) (c) Individual (JOB)

Figure 20: Rule Distribution Across Workloads and Relative Rule Impact on JOB (Gemini-2.5-pro, PostgreSQL).

LLM-based, whole-query rewriting of a workload of queries. Like other LLM-based systems, GenRewrite [60] is a multi-stage rewriting framework. It selects a subset of workload queries as seeds and uses an LLM to generate natural-language rewrite rules, which are then fed back to the model to rewrite queries. These rules are reused for future queries. However, this approach requires per-query LLM interactions, struggles to scale to large workloads, and relies on verification by experts. In contrast, ReSequel leverages metadata to guide template rewriting with caching and verification.

Rewrites in Query Optimization: Beyond heuristic rewrite systems, prior work has integrated rewrites directly into cost-based query optimization. The Cascades framework [37] applies rewrite rules during exploration of group expressions, intertwining rewrite application with cost-based join enumeration. Other approaches integrate specific rewrites into join optimization, such as pre-aggregation before joins [18, 94]. In addition, cost-based optimizers have been extended with specialized physical operators, including group-join [67], which captures common rewrite patterns combining joins and group-by aggregation on the same key. In contrast, we perform holistic rewrites over join orders and related transformations on individual query instances.

Automatic Query Verification: Verifying rewritten queries is essential for correctness. Traditional systems define equivalence using source-target patterns [66] and apply them iteratively. Automatic rewrite discovery systems also rely on verification [92]. SQLSolver [21] determines query equivalence using linear integer arithmetic and existing solvers. Beyond these methods, prior work on SQL testing [78] and text-to-SQL generation [74] compare query results directly. ReSequel follows this result-oriented approach, comparing outputs across rewritten templates on samples.

9 CONCLUSIONS

We introduced ReSequel, an end-to-end query rewriting system that operates on top of existing DBMSs and uses LLMs to optimize groups of related queries. ReSequel combines template-level rewriting with verification and Top-1 query selection. We draw three conclusions. First, despite decades of work on query rewriting, large rule sets and their interactions remain brittle. Phase ordering, limited rewrite budgets, and strict pattern matching prevent DBMSs from fully optimizing messy queries. Second, rewriting recurring query templates, rather than individual queries, substantially speeds up rewriting. This approach enables scalable, reusable, and robust optimization. Third, in a metadata- and statistics-guided manner, LLMs are capable of correct query rewriting that yields substantial runtime improvements on various DBMSs. Future work includes generating DBMS-specific rewrite rules from verified LLM-generated rewrites for integration into existing DBMS code bases.

REFERENCES

[1] 2025. Apache Calcite is a dynamic data management framework. https://calcite.apache.org/

[2] 2025. Groq Cloud. https://console.groq.com/

[3] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The Unreasonable Effectiveness of LLMs for Query Optimization. *NeurIPS* (2024). https://doi.org/10.48550/ARXIV.2411.02862

[4] Morton M. Astrahan et al. 1976. System R: Relational Approach to Database Management. *ACM Trans. Database Syst.* 1, 2 (1976), 97–137. https://doi.org/10.1145/320455.320457

[5] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *PVLDB* 16, 11 (2023), 2911–2924. https://doi.org/10.14778/3611479.3611497

[6] Eirik Bakke and David R. Karger. 2016. Expressive Query Construction through Direct Manipulation of Nested Relational Results. In *SIGMOD*. 1377–1392. https://doi.org/10.1145/2882903.2915210

[7] Zhuowei Bao, Benny Kimelfeld, and Yunyao Li. 2012. Automatic suggestion of query-rewrite rules for enterprise search. In *SIGIR*. 591–600. https://doi.org/10.1145/2348283.2348363

[8] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Adaptive query compilation in graph databases. *Distributed Parallel Databases* 41, 3 (2023), 359–386. https://doi.org/10.1007/S10619-023-07430-4

[9] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *SIGMOD*. 221–230. https://doi.org/10.1145/3183713.3190662

[10] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*. 1–10. https://doi.org/10.1145/223784.223785

[11] Altan Birler, Alfons Kemper, and Thomas Neumann. 2024. Robust Join Processing with Diamond Hardened Joins. *PVLDB* 17, 11 (2024), 3215–3228. https://doi.org/10.14778/3681954.3681995

[12] Matthias Böhm, Douglas R. Burdick, Alexandre V. Evfimievski, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Shirish Tatikonda, and Yuanyuan Tian. 2014. SystemML's Optimizer: Plan Generation for Large-Scale Machine Learning Programs. *Data Eng. Bull.* 37, 3 (2014), 52–62. http://sites.computer.org/debull/A14sept/p52.pdf

[13] Nicolas Bruno and Surajit Chaudhuri. 2006. To Tune or not to Tune? A Lightweight Physical Design Alerter. In *VLDB*. 499–510. http://dl.acm.org/citation.cfm?id=1164171

[14] Sunil Chakkappen, Shreya Kunjibettu, Daniel Mcgreer, Masoomeh Kishi, Hong Su, Mohamed Ziauddin, Mohamed Zaït, Zhan Li, and Yuying Zhang. 2025. Automatic Indexing in Oracle. *PVLDB* 18, 12 (2025), 4924–4937. https://doi.org/10.14778/3750601.3750616

[15] Donald D. Chamberlin, Morton M. Astrahan, Kapali P. Eswaran, Patricia P. Griffiths, Raymond A. Lorie, James W. Mehl, Phyllis Reisner, and Bradford W. Wade. 1976. SEQUEL 2: A Unified Approach to Data Definition, Manipulation, and Control. *IBM J. Res. Dev.* 20, 6 (1976), 560–575. https://doi.org/10.1147/RD.206.0560

[16] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A Structured English Query Language. In *SIGMOD*. 249–264. https://doi.org/10.1145/800296.811515

[17] Konstantinos Chasialis, Yannis Foufoulas, Alkis Simitsis, and Yannis E. Ioannidis. 2026. Optimizing UDF Queries in SQL Data Engines. In *EDBT*. https://doi.org/10.48786/EDBT.2026.21

[18] Surajit Chaudhuri and Kyuseok Shim. 1994. Including Group-By in Query Optimization. In *VLDB*. 354–366. http://www.vldb.org/conf/1994/P354.PDF

[19] Transaction Processing Council. 2025. https://www.tpc.org/tpch.

[20] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *PVLDB* 14, 13 (2021), 3376–3388. https://doi.org/10.14778/3484224.3484234

[21] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proc. ACM Manag. Data* 1, 4 (2023), 227:1–227:26. https://doi.org/10.1145/3626768

[22] Jens Dittrich and Joris Nix. 2020. The Case for Deep Query Optimisation. In *CIDR*. http://cidrdb.org/cidr2020/papers/p3-dittrich-cidr20.pdf

[23] Jens Dittrich, Joris Nix, and Christian Schön. 2021. The next 50 Years in Database Indexing or: The Case for Automatically Generated Index Structures. *PVLDB* 15, 3 (2021), 527–540. https://doi.org/10.14778/3494124.3494136

[24] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. 2023. SlabCity: Whole-Query Optimization Using Program Synthesis. *PVLDB* 16, 11 (2023), 3151–3164. https://doi.org/10.14778/3611479.3611515

[25] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *VLDB* 13, 8 (2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[26] Tarek Elgamal, Shangyu Luo, Matthias Boehm, Alexandre V. Evfimievski, Shirish Tatikonda, Berthold Reinwald, and Prithviraj Sen. 2017. SPOOF: Sum-Product Optimization and Operator Fusion for Large-Scale Machine Learning. In *CIDR*. http://cidrdb.org/cidr2017/papers/p3-elgamal-cidr17.pdf

[27] Jingzhi Fang, Yanyan Shen, Yue Wang, and Lei Chen. 2020. Optimizing DNN Computation Graph using Graph Substitutions. *PVLDB* 13, 11 (2020), 2734–2746. http://www.vldb.org/pvldb/vol13/p2734-fang.pdf

[28] Amela Fejza, Pierre Genevès, and Nabil Layaïda. 2024. Efficient Enumeration of Recursive Plans in Transformation-based Query Optimizers. *PVLDB* 17, 11 (2024), 3095–3108. https://doi.org/10.14778/3681954.3681986

[29] Philipp Fent, Altan Birler, and Thomas Neumann. 2023. Practical planning and execution of groupjoin and nested aggregates. *VLDB J.* 32, 6 (2023), 1165–1190. https://doi.org/10.1007/S00778-022-00765-X

[30] Sofoklis Floratos, Ahmad Ghazal, Jason Sun, Jianjun Chen, and Xiaodong Zhang. 2021. DBSpinner: Making a Case for Iterative Processing in Databases. In *ICDE*. 2399–2410. https://doi.org/10.1109/ICDE51399.2021.00273

[31] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *VLDB* 16, 12 (2023), 3874–3877. https://doi.org/10.14778/3611540.3611574

[32] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *VLDB* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[33] Jonathan Fürst, Catherine Kosten, Farhad Nooralahzadeh, Yi Zhang, and Kurt Stockinger. 2025. Evaluating the Data Model Robustness of Text-to-SQL Systems Based on Real User Queries. In *EDBT*. https://doi.org/10.48786/EDBT.2025.13

[34] Rainer Gemulla, Philipp Rösch, and Wolfgang Lehner. 2008. Linked Bernoulli Synopses: Sampling along Foreign Keys. In *SSDBM*, Vol. 5069. 6–23. https://doi.org/10.1007/978-3-540-69497-7_4

[35] Parke Godfrey, Jarek Gryz, Andrzej Hoppe, Wenbin Ma, and Calisto Zuzarte. 2009. Query Rewrites with Views for XML in DB2. In *ICDE*. 1339–1350. https://doi.org/10.1109/ICDE.2009.131

[36] Google. 2025. https://gemini.google.com.

[37] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[38] Luca Gretscher and Jens Dittrich. 2025. How to Optimize SQL Queries? A Comparison Between Split, Holistic, and Hybrid Approaches. *PVLDB* 18, 11 (2025), 3910–3922. https://doi.org/10.14778/3749646.3749663

[39] CWI Database Architectures Group. 2025. https://github.com/cwida/public_bi_benchmark.

[40] Immanuel Haffner and Jens Dittrich. 2023. A simplified Architecture for Fast, Adaptive Compilation and Execution of SQL Queries. In *EDBT*. https://doi.org/10.48786/EDBT.2023.01

[41] Yuxing Han et al. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *PVLDB* 15, 4 (2021), 752–765. https://doi.org/10.14778/3503585.3503586

[42] Cinda Heeren, H. V. Jagadish, and Leonard Pitt. 2003. Optimal indexing using near-minimal space. In *PODS*. 244–251. https://doi.org/10.1145/773153.773177

[43] Gerald Held, Michael Stonebraker, and Eugene Wong. 1975. INGRES: A Relational Data Base System. In *AFIPS*, Vol. 44. 409–416. https://doi.org/10.1145/1499949.1500029

[44] Moshik Hershcovitch, Artem Khyzha, Daniel G. Waddington, and Adam Morrison. 2022. Elastic Indexes: Dynamic Space vs. Query Efficiency Tuning for In-Memory Database Indexing. In *EDBT*. https://doi.org/10.48786/EDBT.2022.18

[45] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees Over Normalized Data Using Only SQL. *PVLDB* 16, 11 (2023), 3071–3084. https://doi.org/10.14778/3611479.3611509

[46] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *SOSP*. 47–62. https://doi.org/10.1145/3341301.3359630

[47] Zhihao Jia, James Thomas, Todd Warszawski, Mingyu Gao, Matei Zaharia, and Alex Aiken. 2019. Optimizing DNN Computation with Relaxed Graph Substitutions. In *MLSys*. https://proceedings.mlsys.org/paper_files/paper/2019/hash/

[48] David Justen, Daniel Ritter, Campbell Fraser, Andrew Lamb, Nga Tran, Allison Lee, Thomas Bodner, Mhd Yamen Haddad, Steffen Zeuch, Volker Markl, and Matthias Boehm. 2024. POLAR: Adaptive and Non-invasive Join Order Selection via Plans of Least Resistance. *PVLDB* 17, 6 (2024), 1350–1363. https://doi.org/10.14778/3648160.3648175

[49] George Katsogiannis-Meimarakis and Georgia Koutrika. 2021. A Deep Dive into Deep Learning Approaches for Text-to-SQL Systems. In *SIGMOD*. 2846–2851. https://doi.org/10.1145/3448016.3457543

[50] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *SIGMOD*. 715–730. https://doi.org/10.1145/3035918.3064049

[51] Steffen Kläbe and Kai-Uwe Sattler. 2023. Patched Multi-Key Partitioning for Robust Query Performance. In *EDBT*. https://doi.org/10.48786/EDBT.2023.26

[52] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2023. Correction to: Data dependencies for query optimization: a survey. *VLDB J.* 32, 2 (2023), 471. https://doi.org/10.1007/S00778-021-00710-4

[53] Muralidhar Krishnaprasad, Zhen Hua Liu, Anand Manikutty, James W. Warner, Vikas Arora, and Susan Kotsovolos. 2004. Query Rewrite for XML in Oracle XML DB. In *VLDB*. 1122–1133. https://doi.org/10.1016/B978-012088469-8.50098-X

[54] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *PVLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[55] Viktor Leis, Kan Kundhikanjana, Alfons Kemper, and Thomas Neumann. 2015. Efficient Processing of Window Functions in Analytical SQL Queries. *PVLDB* 8, 10 (2015), 1058–1069. https://doi.org/10.14778/2794367.2794375

[56] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*. http://cidrdb.org/cidr2017/papers/p9-leis-cidr17.pdf

[57] Viktor Leis, Bernhard Radke, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2018. Query optimization through the looking glass, and what we found running the Join Order Benchmark. *VLDB J.* 27, 5 (2018), 643–668. https://doi.org/10.1007/S00778-017-0480-7

[58] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-Based Rewrite System for Boosting Query Efficiency. *PVLDB* 18, 1 (2024), 53–65. https://doi.org/10.14778/3696435.3696440

[59] Daniel Lindner, Daniel Ritter, and Felix Naumann. 2024. Enabling Data Dependency-based Query Optimization. *CoRR* abs/2406.06886 (2024). https://doi.org/10.48550/ARXIV.2406.06886

[60] Jie Liu and Barzan Mozafari. 2026. GenRewrite: Query Rewriting via Large Language Models. *SIGMOD* 4, 1 (2026), 1–26. https://dl.acm.org/doi/pdf/10.1145/3786684

[61] Ester Livshits, Rina Kochirgan, Segev Tsur, Ihab F. Ilyas, Benny Kimelfeld, and Sudeepa Roy. 2021. Properties of Inconsistency Measures for Databases. In *SIGMOD*. 1182–1194. https://doi.org/10.1145/3448016.3457310

[62] Guy Lohman. 2002. The DB2 Universal Database Optimizer. Guest Lecture. https://cs.uwaterloo.ca/~ilyas/CS448W14/ibm.pdf.

[63] Guy Lohman. 2014. Is Query Optimization a "Solved" Problem? ACM SIGMOD Blog. https://wp.sigmod.org/?p=1075.

[64] Toby Mao. 2024. SQLGlot. GitHub repository. Retrieved April 29, 2026 from https://github.com/tobymao/sqlglot.

[65] Volker Markl, Vijayshankar Raman, David E. Simmen, Guy M. Lohman, and Hamid Pirahesh. 2004. Robust Query Processing through Progressive Optimization. In *SIGMOD*. 659–670. https://doi.org/10.1145/1007568.1007642

[66] Guido Moerkotte. 2025. Building Query Compilers. https://pi3.informatik.uni-mannheim.de/~moer/querycompiler.pdf Last Accessed: October 03, 2025.

[67] Guido Moerkotte and Thomas Neumann. 2011. Accelerating Queries with Group-By and Join by Groupjoin. *PVLDB* 4, 11 (2011), 843–851. http://www.vldb.org/pvldb/vol4/p843-moerkotte.pdf

[68] Magnus Müller, Lucas Woltmann, and Wolfgang Lehner. 2023. Enhanced Featurization of Queries with Mixed Combinations of Predicates for ML-based Cardinality Estimation. In *EDBT*. https://doi.org/10.48786/EDBT.2023.22

[69] Parimarjan Negi, Ryan Marcus, Andreas Kipf, Hongzi Mao, Nesime Tatbul, Tim Kraska, and Mohammad Alizadeh. 2021. Flow-Loss: Learning Cardinality Estimates That Matter. *PVLDB* 14, 11 (jul 2021), 2019–2032. https://doi.org/10.14778/3476249.3476259

[70] Thomas Neumann and Alfons Kemper. 2015. Unnesting Arbitrary Queries. In *BTW*, Vol. P-241. 383–402. https://dl.gi.de/handle/20.500.12116/2418

[71] Thomas Neumann and Viktor Leis. 2024. A Critique of Modern SQL and a Proposal Towards a Simple and Expressive Query Language. In *CIDR*. https://www.cidrdb.org/cidr2024/papers/p48-neumann.pdf

[72] Oleksii Vasyliev. 2025. https://pgtune.leopard.in.ua.

[73] Amy Ousterhout, Steven McCanne, Henri Dubois-Ferrière, Silvery Fu, Sylvia Ratnasamy, and Noah Treuhaft. 2023. Zed: Leveraging Data Types to Process Eclectic Data. In *CIDR*. https://www.cidrdb.org/cidr2023/papers/p52-ousterhout.pdf

[74] Simone Papicchio, Paolo Papotti, and Luca Cagliero. 2023. QATCH: Benchmarking SQL-centric tasks with Table Representation Learning Models on Your Data. In *NeurIPS*. http://papers.nips.cc/paper_files/paper/2023/hash/62a24b69b820d30e9e5ad4f15ff7bf72-Abstract-Datasets_and_Benchmarks.html

[75] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*. 39–48. https://doi.org/10.1145/130283.130294

[76] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. 1997. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *ICDE*. 391–400. https://doi.org/10.1109/ICDE.1997.581945

[77] PostgreSQL. 2026. CREATE CAST. https://www.postgresql.org/docs/current/sql-createcast.html Last Accessed: January 01, 2026.

[78] Manuel Rigger and Zhendong Su. 2020. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE*. 1140–1152. https://doi.org/10.1145/3368089.3409710

[79] Dario Satriani, Enzo Veltri, Donatello Santoro, Sara Rosato, Simone Varriale, and Paolo Papotti. 2025. Logical and Physical Optimizations for SQL Query Execution over Large Language Models. *Proc. ACM Manag. Data* 3, 3 (2025), 181:1–181:28. https://doi.org/10.1145/3725411

[80] Tobias Schmidt, Philipp Fent, and Thomas Neumann. 2022. Efficiently Compiling Dynamic Code for Adaptive Query Processing. 11–22. http://www.adms-conf.org/2022-camera-ready/ADMS22_schmidt.pdf

[81] Tobias Schmidt, Viktor Leis, Peter A. Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *PVLDB* 18, 11 (2025), 4144–4157. https://www.vldb.org/pvldb/vol18/p4144-schmidt.pdf

[82] Pranjal Shankhdhar, Feilong Liu, Jay Narale, James Sun, Rebecca Schlussel, and Lyublena Antova. 2024. Presto's History-based Query Optimizer. *PVLDB* 17, 12 (2024), 4077–4089. https://doi.org/10.14778/3685800.3685828

[83] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *PVLDB* 15, 5 (2022), 1119–1131. https://doi.org/10.14778/3510397.3510408

[84] Tarique Siddiqui and Wentao Wu. 2023. ML-Powered Index Tuning: An Overview of Recent Progress and Open Challenges. *SIGMOD Rec.* 52, 4 (2023), 19–30. https://doi.org/10.1145/3641832.3641836

[85] Jiansen Song, Wensheng Dou, Yu Gao, Ziyu Cui, Yingying Zheng, Dong Wang, Wei Wang, Jun Wei, and Tao Huang. 2024. Detecting Metadata-Related Logic Bugs in Database Systems via Raw Database Construction. *PVLDB* 17, 8 (2024), 1884–1897. https://doi.org/10.14778/3659437.3659445

[86] Zhaoyan Sun, Xuanhe Zhou, Guoliang Li, Xiang Yu, Jianhua Feng, and Yong Zhang. 2025. R-Bot: An LLM-based Query Rewrite System. *PVLDB* 18, 12 (2025), 5031–5044. https://www.vldb.org/pvldb/vol18/p5031-li.pdf

[87] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. 2025. Can Large Language Models Be Query Optimizer for Relational Databases? *arXiv preprint arXiv:2502.05562* (2025). https://arxiv.org/pdf/2502.05562

[88] Nikolaos Tziavelis, Wolfgang Gatterbauer, and Mirek Riedewald. 2020. Optimal Join Algorithms Meet Top-k. In *SIGMOD*. 2659–2665. https://doi.org/10.1145/3318464.3383132

[89] Adrian Vogelsgesang, Tobias Mühlbauer, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Domain Query Optimization: Adapting the General-Purpose Database System Hyper for Tableau Workloads. In *BTW (LNI)*, Vol. P-289. 313–333. https://doi.org/10.18420/BTW2019-19

[90] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. 2020. SPORES: Sum-Product Optimization via Relational Equality Saturation for Large Scale Linear Algebra. *PVLDB* 13, 11 (2020), 1919–1932. http://www.vldb.org/pvldb/vol13/p1919-wang.pdf

[91] Zijia Wang, Haoran Liu, Chen Lin, Zhifeng Bao, Guoliang Li, and Tianqing Wang. 2024. Leveraging Dynamic and Heterogeneous Workload Knowledge to Boost the Performance of Index Advisors. *PVLDB* 17, 7 (2024), 1642–1654. https://doi.org/10.14778/3654621.3654631

[92] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD*. 94–107. https://doi.org/10.1145/3514221.3526125

[93] Marianne Winslett. 2003. Interview with Pat Selinger. *SIGMOD Rec.* 32, 4 (2003), 93–103. http://www.acm.org/sigmod/record/issues/0312/17.selinger-interview.pdf

[94] Weipeng P. Yan and Per-Åke Larson. 1995. Eager Aggregation and Lazy Aggregation. In *VLDB*. 345–357. http://www.vldb.org/conf/1995/P345.PDF

[95] Zhengtong Yan, Valter Uotila, and Jiaheng Lu. 2023. Join Order Selection with Deep Reinforcement Learning: Fundamentals, Techniques, and Challenges. *PVLDB* 16, 12 (2023), 3882–3885. https://doi.org/10.14778/3611540.3611576

[96] Weijie Zhao, Florin Rusu, Bin Dong, and Kesheng Wu. 2016. Similarity Join over Array Data. 2007–2022. https://doi.org/10.1145/2882903.2915247

[97] Wenchao Zhou, Yifan Cai, Yanqing Peng, Sheng Wang, Ke Ma, and Feifei Li. 2021. VeriDB: An SGX-based Verifiable Database. In *SIGMOD*. 2182–2194. https://doi.org/10.1145/3448016.3457308

[98] Wei Zhou, Chen Lin, Xuanhe Zhou, Guoliang Li, and Tianqing Wang. 2024. TRAP: Tailored Robustness Assessment for Index Advisors via Adversarial Perturbation. In *ICDE*. 42–55. https://doi.org/10.1109/ICDE60146.2024.00011

[99] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using Monte Carlo tree search. *PVLDB* 15, 1 (2021), 46–58. https://doi.org/10.14778/3485450.3485456

[100] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. Db-gpt: Large language model meets database. *Data Science and Engineering* 9, 1 (2024), 102–111. https://doi.org/10.1007/s41019-023-00235-6