PaperID: 1378

LITHE: Robust Query Transformation using LLMs

Sriram Dharwada*
Indian Institute of Science
sriramd@iisc.ac.in

Himanshu Devrani*
Indian Institute of Science
himanshud@iisc.ac.in

Jayant R Haritsa
Indian Institute of Science
haritsa@iisc.ac.in

Harish Doraiswamy
Microsoft Research India
harish.doraiswamy@microsoft.com

ABSTRACT

Query rewriting is a classical technique for transforming complex declarative SQL queries into "lean" equivalents that are conducive to (a) faster execution from a performance perspective, and (b) better understanding from a developer perspective. The rewriting is typically achieved via transformation rules, but these rules are limited in scope and difficult to update in a production system. In recent times, LLM-based techniques have also been mooted, but they are prone to both semantic and syntactic errors.

We investigate here, how the remarkable cognitive capabilities of LLMs can be leveraged for performant query rewriting while incorporating safeguards and optimizations to ensure correctness and efficiency. Our study shows that these goals can be progressively achieved through incorporation of (a) an ensemble suite of basic prompts, (b) database-sensitive prompts via redundancy removal and selectivity-based rewriting rules, and (c) LLM token probability-guided rewrite paths. Further, a suite of statistical and logic-based tools can be used to guard against errors produced by the model.

We have implemented the above LLM-infused techniques in the `LITHE` system, and evaluated complex analytic queries from multiple benchmarks on contemporary database platforms. The results show significant improvements over SOTA rewriting techniques – for instance, on TPC-DS, `LITHE` constructed productive (>1.5x speedup) rewrites for *two-thirds* of the query suite, delivering four times more coverage than SOTA. Further, the geometric mean of its estimated execution speedups was an *order-of-magnitude* jump over SOTA performance. In essence, `LITHE` offers a potent and robust LLM-based intermediary between enterprise applications and database engines.

PVLDB Reference Format:
Sriram Dharwada, Himanshu Devrani, Jayant R Haritsa, and Harish Doraiswamy. LITHE: Robust Query Transformation using LLMs. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at URL_TO_YOUR_ARTIFACTS.

*Equal contributions.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

The SQL queries embedded in enterprise applications are often riddled with inefficiencies and redundancies, especially when machine-generated by modeling software such as ORM tools. A compelling case in point is the blog-processing query [4] shown in Figure 1 – this query was produced by the popular Entity Framework [21] mapper. It computes a daily summary of rating metrics for highly-rated blogs. This seemingly complex nested query can be equivalently rewritten in the "lean" flat version shown in Figure 2.

```sql
SELECT t.Key, sum(t.Rating) AS PostRating,
   (SELECT sum(b0.Rating)
    FROM (SELECT p0.PostId, p0.BlogId, p0.Content,
          p0.CreatedDate, p0.Rating, p0.Title,
          b1.BlogId AS BlogId0, b1.Rating AS Rating0,
          b1.Url, p0.day AS Key
          FROM Posts AS p0 INNER JOIN Blogs AS b1 ON
              p0.BlogId = b1.BlogId
          WHERE b1.Rating > 5) AS t0
    INNER JOIN Blogs AS b0 ON t0.BlogId = b0.BlogId
    WHERE t.Key = t0.Key ) AS BlogRating
FROM (SELECT p.Rating, p.day AS Key
      FROM Posts AS p INNER JOIN Blogs AS b
         ON p.BlogId = b.BlogId
      WHERE b.Rating > 5) AS t
GROUP BY t.Key;
```

Figure 1: Complex SQL Representation

```sql
SELECT p.day AS Key, SUM(p.Rating) AS PostRating,
       SUM(b.Rating) AS BlogRating
FROM Posts AS p INNER JOIN Blogs AS b
       ON p.BlogId = b.BlogId
WHERE b.Rating > 5
GROUP BY p.day;
```

Figure 2: Lean Equivalent Query

Albeit less common, we observe such superfluity even in the hand-crafted expert query domain – for instance, TPC-DS Query Q23 includes a sub-query that identifies customers with the maximum store sales in a four-year period; this sub-query has a redundant join of STORE_SALES with CUSTOMER despite the FK-PK relationship between these tables, as highlighted in [25].

Transforming bloated queries to lean equivalents in the application source code is beneficial from multiple perspectives: First, rewrites can improve code readability, a major concern in industrial contexts. Second, query optimizers are expected to, in principle, remove bloat while constructing efficient execution plans. However, in practice, they are often led astray by complex representations, resulting in poor query performance. In fact, both examples discussed above were implemented as such by the current PostgreSQL optimizer (v16) [1], without any change. Therefore, a query rewriter can serve as an effective and non-invasive mechanism for delivering good performance despite inherent optimizer limitations.

Given this motivational context, a viable SQL-to-SQL query transformer should satisfy the following criteria: (1) The transformed query should be semantically equivalent to the original; (2) There should ideally be performance improvement due to the rewrite, but at the very least, no regression in query performance; and (3) The transformation overheads must be practical for deployment.

Prior Work

A variety of innovative *Rule-based* (e.g. Learned Rewrite [38]) and *Model-based* (e.g. Gen-Rewrite [19]) approaches have been proposed in the literature for SQL query transformations. However, as explained later in the related work section (Section 8), the benefits of these state-of-the-art (SOTA) techniques are curtailed by: (a) restrictions in rewrite scope, (b) susceptibility to semantic and syntactic errors, and (c) transformations via the plan space rather than directly in query space.

An alternative and extremely simple strategy would be to take advantage of the large body of recent literature on LLM-based Text-to-SQL translations (e.g. [17, 33, 35]). That is, first invoke an LLM to transform the original SQL to text, and then employ a Text-to-SQL tool to produce quality SQL. However, as discussed later in Section 7, we found this approach to be highly prone to semantic errors and lacking execution benefits. This is perhaps to be expected given the focus of Text-to-SQL tools on query formulation, orthogonal to our objective of performance enhancement. Therefore, the SQL-to-SQL problem merits a fresh assessment from first principles.

The `LITHE` Rewriter

In this study, we carry out a calibrated investigation of how LLMs could be reliably used to produce performant rewrites of complex SQL queries. The proposed techniques, described below, have been implemented and evaluated in the `LITHE` (LLM Infused Transformations of Hefty queries) rewriting system developed by our group.

Basic Prompts. We begin with an ensemble suite of generic prompts that cover a spectrum of detail, ranging from a single summary sentence to detailed instructions running to several paragraphs. Interestingly, we find that more information is not necessarily better wrt rewriting quality, and the best prompt granularity is query-specific. Moreover, this simple prompt ensemble was found to deliver performance similar to the `SOTA` techniques.

Database-sensitive Prompts. To help the model adapt to different query patterns and structures, we next introduce rules in the prompts. But our choice and usage of rules are markedly different from those used in prior work. First, our rules are invoked directly in *query space*. The LLM therefore gains the latitude to generalize the rule usage to a wide range of queries. This is in contrast to the hardwired and narrow rule application mechanisms (e.g. Calcite [8] rules) typically used in existing rewrite systems, which operate in *plan space*. Second, to minimize rule application overheads, we incorporate only a handful of rules that serve to eliminate the primary causes of poor query performance. Whereas current rewrite systems employ a large number of rules in their processing. In particular, we work with two classes of rules:

1. Redundancy Removal Rules: These rules are meant to eliminate repeated and redundant computations of the same output.

2. Metadata-infused Rules: These rules make use of the rich metadata available in database environments, such as the logical schema (table definitions and constraints) and predicate selectivities, and include this information in the LLM prompts. To our knowledge, such metadata inclusion has not been considered before in the SQL rewrite context. As shown later in our experiments, it proves to be a powerful mechanism for ensuring performant rewrites across database environments.

Token Probability Driven Rewrites. The above rewrite options are restricted to the standard prompt interface. Additionally, we leverage the rich telemetry provided by LLMs – in particular, the *token probabilities* output at each step in the prediction sequence. Whenever the LLM lacks high confidence in the next token, we follow multiple alternative paths in the decision process. To ensure practical overheads on this enumerative approach, a Monte Carlo tree search (MCTS) technique is incorporated in our implementation. MCTS has previously demonstrated significant effectiveness in code generation exercises [36].

Experimental Evaluation

Our experiments are carried out on a variety of synthetic and real benchmarks: TPC-DS [11], DSB [12], JOB [15], StackOverflow [20] and Archer [37], covering a broad spectrum of databases and queries. Further, anonymized versions, where the table and column names are altered to not convey any semantic meaning about their contents, are also evaluated to assess robustness to databases not seen by the LLM during its training phase. These databases are hosted on PostgreSQL and GPT-4o is used as the LLM.

We compare the performance of `LITHE` against `SOTA` techniques (specifically, Learned Rewrite [38], LLM-R^2 [18], GenRewrite [19]) as well as a baseline LLM prompt [19]. The primary metrics are the execution cost reductions and the rewriting overheads. For LLM-based techniques, the number of tokens in the prompts is also monitored since the financial charges for LLM usage are typically dependent on this number. To understand the impact of the various components of `LITHE`, a systematic ablation study is carried out. Finally, the performance on alternative database and LLM platforms is also evaluated.

Verifying Semantic Equivalence. The general problem of proving query equivalence has long been known to be computationally hard [6]. However, for restricted classes of queries, various logic-based equivalence-proving tools are available (e.g. Cosette [10], SQL-Solver [13], and QED [30]). In our `LITHE` system, we use QED [30] to verify rewrite correctness for queries within its scope. The remainder are checked, albeit not provably, by computing result equivalences on a diverse cluster of databases using small sampled versions of the original database. We also provide the user with an option to test result equivalence using the original database.

Results. Our experiments demonstrate that LITHE achieves semantically correct transformations that deliver a significant reduction in estimated execution times for many queries. Moreover, LITHE performed better than or at least as well as SOTA on *all* the benchmarks. As a compelling case in point, LITHE constructed "productive" (>1.5x speedup) rewrites for about *two-thirds* of the complete TPC-DS query suite, delivering four times more coverage than SOTA techniques. Further, the GM (Geometric Mean) of the reduction in query execution costs was **30.6**, an *order-of-magnitude* jump over that offered by SOTA.

In essence, LITHE offers a potent LLM-based intermediary between enterprise applications and database engines.

Contributions

In summary, our study makes the following contributions:

(1) Assesses LLM suitability for SQL-to-SQL transformation.
(2) Transforms directly in query space instead of a plan space intermediate representation, leading to robust rewrites.
(3) Incorporates potent database-sensitive rules in LLM prompts covering schematic and statistical dimensions.
(4) Leverages LLM token probabilities to guide navigation of the rewrite search space and minimize LLM errors.
(5) Evaluates rewriting robustness over a broad range of database environments and benchmarks.
(6) Demonstrates substantive performance benefits, well beyond SOTA, from LLM-powered query rewrites.

2 LITHE OVERVIEW

The LITHE architecture, illustrated in Figure 3, consists of two main components: A prompt-based rewriting pipeline, and a token probability-driven rewrite exploration.

Prompt-based rewriting pipeline. This pipeline comprises four modules: (1) LLM Prompting, (2) Syntax Verification, (3) Query Costing, and (4) Semantic Verification. The user query and an initial prompt are fed to the LLM prompting module, requesting a rewrite. The LLM output is checked by the syntax verification component – if invalid, the error is fed back to the LLM via an updated prompt asking for a correction, and this goes on iteratively until a valid SQL query is obtained. Then, the execution cost of this version is evaluated – if more than the cost of the original query, the rewrite is discarded. Otherwise, the rewrite is tested for semantic equivalence. If a match is established, the rewrite is returned; otherwise, the original query itself is returned. Finally, to prevent runaway situations, a threshold is set on the number of syntax correction attempts (5 in our experiments) and the original query is returned if this threshold is breached.

The above process is repeated for each of the different prompts described in Sections 3 and 4. Then, the prompt providing the least-cost rewrite is returned. Since each prompt is evaluated within a fresh LLM context, there is no dependency on the order in which they are assessed.

Within the pipeline, LITHE uses (a) the query parser from the database engine for syntax verification, (b) the query optimizer for cost estimation, and (c) the techniques described in Section 6.3 to check semantic equivalence.

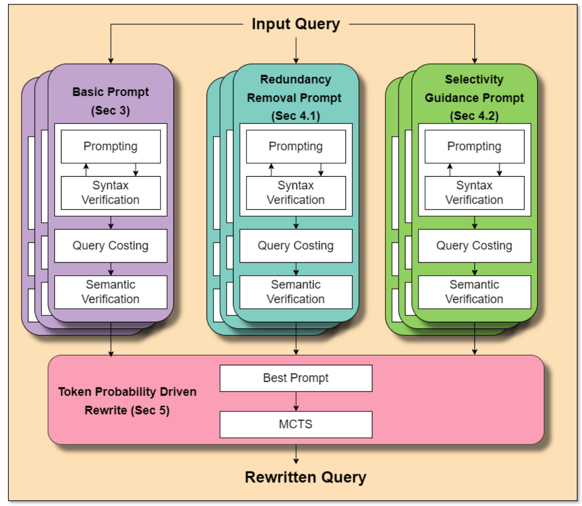

Figure 3: High-level architecture of LITHE

Token probability-driven rewrite. The pipeline-chosen prompt is used as an input to a Monte-Carlo tree search (MCTS)-based procedure to further improve the query rewrite quality. Specifically, multiple exploration paths are followed whenever the LLM lacks high confidence in the predicted token (details in Section 5).

2.1 Performance Framework

As mentioned in the Introduction, the majority of our experiments are processed on GPT-4o, the popular OpenAI LLM, and the rewrites are evaluated on the PostgreSQL v16 database engine.

We define a **productive rewrite (PR)** as one that improves the query's performance by at least **1.5** times wrt the execution times estimated by the database engine's query optimizer – this aggressive choice of threshold is to make the rewrite outcome desirable from an industrial perspective. Given a query workload, the set of *feasible productive rewrites (FPR)* consists of the queries for which PR are known to be feasible by *some* technique or the other. Note that it may be inherently impossible to produce PR on queries that are structurally simple or efficiently written by human experts. Finally, a *maximum productive rewrite (MPR)* is when the PR speedup of the rewrite matches the best known PR on this query.

The overall benefit provided by a rewriting tool is quantified by the number of PR obtained on the query workload. Additionally, we also measure **SpeedupGM**, the *geometric mean* (GM) of the speedups obtained on the FPR queries in this workload. (The geometric mean is used to ensure equal weight for all queries in the workload, irrespective of their relative durations.)

2.2 Query Micro-benchmark

To motivate the progression of strategies incorporated in LITHE, we create an initial micro-benchmark comprising 10 diverse TPC-DS queries for which we were able to hand-craft productive rewrites. These human rewrites deliver a SpeedupGM of **13.9** on the micro-benchmark, serving as an informal "upper-bound" on what is computationally attainable. Later, in Section 7, we extend the evaluation to a suite of complete benchmarks.

Prompt 1

> **Enter Query here**
>
> Rewrite this query to improve performance.

Prompt 2

> You are a database expert and SQL optimizer. Your role involves identifying inefficient SQL queries and transforming them into optimized, functionally equivalent <Database Engine> versions.
>
> **Enter Query here**
>
> Rewrite this query to improve performance of query while maintaining semantic equivalence.

Prompt 3 / Prompt 4

> You are a database expert and SQL optimizer. Your role involves identifying inefficient SQL queries and transforming them into optimized, functionally equivalent <Database Engine> versions. Your tone is analytical and instructional.
>
> A user has provided the following <Database Engine> query that is potentially inefficient:
>
> **Enter Query here**
>
> The task is to first identify whether the query is inefficient or not. If it is inefficient, you must rewrite the query to make it more efficient while maintaining semantic equivalence. Here are a few steps that can help you complete the task:
>
> 1. Start by identifying specific inefficiencies in the provided query. If you feel the original query is already efficient, skip the next two steps, and simply return the query as-is.
>
> 2. Next, provide guidelines for optimizations, and explain the rationale behind the recommended optimizations. Correspond how these changes would map onto the original query to maintain syntactic and semantic equivalence.
>
> 3. Finally craft the new, optimized <Database Engine> query which includes all the enhancements discussed. Give complete rewritten query with no manual involvement by user.

Figure 4: Templates used for Basic Prompts

3 BASIC PROMPTS

In this section, we explore the simplest interface to LLMs, namely *prompting*, for query rewriting. We evaluate four basic prompts, illustrated in Figure 4, which cover a progressive range of detail in the instructions and test the effectiveness of the LLM's base knowledge.

Prompt 1: This is the baseline prompt used in [19], which simply asks the LLM to rewrite a given query to improve performance.

Prompt 2: Explicit instructions are included to maintain semantic and functional equivalence while rewriting.

Prompt 3: Verbose instructions are given to rewrite the query, providing step-by-step guidance to the LLM to think rationally. It is first asked to pick out potential inefficiencies in the input query, and then tasked to identify approaches to address these inefficiencies. Finally, it is instructed to apply the identified solution. Essentially, the prompt tries to make the LLM reason akin to a human expert.

Prompt 4: The sequence of instructions in Prompt 3 is split into sub-prompts, and provided to the LLM in an *iterative* manner instead of all at once. The idea is to break down the complex instructions given in Prompt 3 into digestible steps that help the LLM focus on individual tasks.

Performance. The performance of the four prompt templates on the micro-benchmark (where all 10 queries are in FPR) is shown in Table 1. We find that less than half the rewrites are productive with individual prompts. However, a drill-down shows that each prompt does the best on certain queries – this opens up the possibility of using all four prompts in parallel, and then choosing the best among them. While this ensemble approach (Row 5 in Table 1) raises the PR to 6, there remain four queries that could not be productively rewritten by the prompts alone.

Table 1: Performance of Basic Prompts

Prompt	# PR	SpeedupGM
Prompt 1	3	2.7
Prompt 2	4	1.57
Prompt 3	3	1.37
Prompt 4	2	1.27
Prompt Ensemble	6	3.6
SOTA Ensemble	5	3.4

The SpeedupGM, shown in the last column, for each individual prompt is less than **3**, while that of the ensemble reaches **3.6**. But these speedups, while productive, are all lower than those of the human rewrites, that is, they are not MPR.

Finally, as another baseline, an ensemble of SOTA techniques (described in Section 7) was also processed on the same platform. They delivered (Row 6 in Table 1) 5 productive rewrites with a GM of 3.4, indicating the wide gap between the current state-of-the-art and what is possible.

4 DATABASE-SENSITIVE PROMPTS

As discussed above, basic prompting needs to be improved on two fronts: (1) Ensuring productive rewrites where feasible; and (2) Maximizing the impact of these productive rewrites. To address these issues, we incorporate database domain knowledge. Specifically, we design a *one shot*-based prompting template, augmented with a set of database-aware rewrite rules. The rules are based on common practices followed by DBAs that are widely applicable, and augmented with precise instructions and useful examples to help guide the LLM in the rewriting process.

As a proof of concept, we explore two categories of rewrites here: (a) Rules that eliminate redundancy in the input queries; and (b) predicate selectivity-based rules that implicitly guide, via query space reformulations, the query optimizer towards efficient query execution plans. Of course, this basic set of rules can be expanded further, but as shown by our experiments, even this minimal set is capable of delivering substantive improvements over a broad set of database environments.

4.1 Redundancy Removal

There are different types of redundancy that can occur in a SQL query – repeated computations, superfluous filter predicates, unnecessary joins, etc. Rules **R1 through R4** in Table 2 are designed to tackle such redundancies. For instance, the redundant PK-FK join highlighted in the Introduction is addressed by rule R4 in Table 2 (the relevant schematic information is also provided in the prompt).

Table 2: Rules for Database-sensitive prompts

	Redundancy Removal Rules
R1	Use CTEs (Common Table Expressions) to avoid repeated computation.
R2	When multiple subqueries use the same base table, rewrite to scan the base table only once.
R3	Remove redundant conjunctive filter predicates.
R4	Remove redundant key (PK-FK) joins.
	Statistics-based Rules
R5	Choose EXIST or IN from subquery selectivity (high/low).
R6	Pre-filter fact tables in a CTE using dimension tables with low selectivities. Retain dimension table filters in main query. Do not create explicit join statements.

The template for such rule-based prompts is shown in Figure 5(a) and includes an example to demonstrate the rule application to the LLM – the specific examples used with our rules are enumerated in the Appendix 9. Note that this prompt template allows for only a *single* rule to be present in the prompt. This was a conscious design choice because LLMs are often overwhelmed by excessive information given in monolithic form. Therefore, we apply each rule using a separate prompt, finally returning the rewrite providing the best performance improvement.

Performance. The performance improvement achieved on the micro-benchmark by an ensemble that adds the redundancy-removing prompts to the basic set (Section 3) is shown in Table 3. We observe that the PR increases to **8**, and SpeedupGM grows to **4.8**.

Table 3: Performance with Redundancy Removal Rules

Prompt	# PR	SpeedupGM
Basic Prompts $\bigcup$ {R1, . . ., R4}	8	4.8

A natural question here would be whether, while retaining the one-rule-per-prompt design, the rules could be *progressively* applied with the output of one prompt provided as input to the next, and so on. This approach would benefit queries with multiple types of redundancies. However, it also increases rewrite overheads due to repeated prompt applications. So, for simplicity, we have chosen to process them individually rather than cumulatively.

4.2 Selectivity-based Guidance

We now turn our attention to rules whose applicability to a query is conditional on the specific database environment, specifically its statistical aspects. For example, consider the alternative rewrites shown in Figure 6 using the EXIST and IN clauses (highlighted in red), respectively – here the appropriate choice is dictated by the *selectivity* of the inner subquery – EXISTS for high selectivity values and IN for low values. Based on such arguments, rules **R5** and **R6**, which pre-filters fact tables for dimension tables with low selectivity values, are included in Table 2. Note that, a specific instruction to not create explicit joins had to be added in rule R6. This is because in the presence of CTEs, the LLM is prone to schematic confusion regarding which attribute belongs to which table, leading it to construct invalid joins.

(a) Redundancy-Removal Prompt

You are a database expert and SQL optimizer. Your role involves identifying inefficient SQL queries and transforming them into optimized, functionally equivalent <Database Engine> versions. Your tone is analytical and instructional.

This is a task to rewrite queries to improve performance using the following rewrite rule:

****Enter Rewrite Rule here****

[ORIGINAL] ****Enter Example Original Query here****

[REWRITTEN] ****Enter Example Rewritten Query here****

Now consider the query below and try to rewrite it to improve the performance.

[ORIGINAL] ****Enter User Input Query here****

[REWRITTEN]

(b) Statistics Guideance Prompt

You are a database optimizer and your task is to analyze the given <Database Engine> query using database schema and statistical information available to you. Find inefficiency in the query. If query is already efficient then return the original query. Otherwise, rewrite the query in a more efficient way to improve its performance, and explain your rewrites.

This is a task to improve query performance using the following rewrite rule:

****Enter Rewrite Rule here****

Here is an example to help you

[ORIGINAL] ****Enter Example Original Query here****

[STATISTICS] ****Enter Statistics for example Query here****

[REWRITTEN] ****Enter Example Rewritten Query here****

Now consider the query and statistics given below and try to rewrite the query to improve performance.

[ORIGINAL] ****Enter User Input Query here****

[Statistics] ****Enter Input Query statistics here****

[REWRITTEN]

Figure 5: Templates for Database-sensitive Prompts

The input prompt for these rules, as shown in Figure 5(b), is modified to include the following:

(1) Selectivities of columns appearing in `WHERE` and `JOIN` clauses, as obtained from the database engine.
(2) Clause rewrite rules and instructions based on statistics.
(3) Examples relevant to the chosen rewrite rules.

Performance. The performance improvements following addition of selectivity-guided prompts are shown in Table 4. We observe that PR are now obtained for *all* 10 micro-benchmark queries. Moreover, the resulting SpeedupGM increases to **11.8**, quite close to the human upper bound of 13.9.

Table 4: Performance of Metadata-infused Prompts

Prompt	# PR	SpeedupGM
Basic Prompts $\bigcup$ {R1, . . ., R6}	10	11.8

A key point to note in closing is that rules R1 through R6 not only bring additional queries into the productive category, but also deliver greater improvement for those already deemed to be productive via the standard prompts of Section 3. Further, to minimize selection overheads, a classifier could be designed to choose the appropriate rule, and this option is examined in Section 7.3.3.

```sql
SELECT c_birth_country, count(*) cnt1
FROM customer c
WHERE (EXISTS (
       SELECT * FROM web_sales, date_dim
       WHERE c.c_customer_sk = ws_bill_customer_sk
         and ws_sold_date_sk = d_date_sk
         and d_year >= 1999 )
    or EXISTS (
       SELECT * FROM catalog_sales, date_dim
       WHERE c.c_customer_sk = cs_ship_customer_sk
         and cs_sold_date_sk = d_date_sk
         and d_year >= 1999 )
  )
GROUP BY c_birth_country;
```

Use EXIST clause when subquery selectivity is high

```sql
SELECT c_birth_country, count(*) cnt1
FROM customer c
WHERE ( c_customer_sk IN (
      SELECT ws_bill_customer_sk
      FROM web_sales, date_dim
      WHERE ws_sold_date_sk = d_date_sk
        and d_year = 1999)
   OR c_customer_sk IN (
      SELECT cs_ship_customer_sk
      FROM catalog_sales, date_dim
      WHERE cs_sold_date_sk = d_date_sk
        and d_year = 1999 )
  )
GROUP BY c_birth_country;
```

Use IN clause when subquery selectivity is low

Figure 6: Example Queries illustrating Rule 5

5 TOKEN PROBABILITY DRIVEN REWRITE

A key challenge when using LLMs is "hallucinations" – the generation of responses that range from being mildly incorrect to completely made up. This is often due to the potential output tokens having low confidence, which in turn results in "confusing" the LLM and generating suboptimal outputs. In order to have a robust approach for such cases, we take inspiration from the code generation literature [36]. Specifically, we propose a *Monte Carlo Tree Search* (MCTS) based decoding approach to search for a sequence of LLM-generated tokens that results in both a valid query rewrite as well as performance improvements.

This approach models the problem of query rewriting as a decision tree denoting a *Markov Decision Process (MDP)* [24]. The root node of the tree corresponds to the initial prompt. An edge from a parent node to a child represents a possible token generated by the LLM and is associated with a value denoting the probability of generating this token given the path taken thus far. Here, each edge can be considered as an *action* of the MDP. A node is considered *terminal* if the incoming edge corresponds to the ";" token, signalling end of the textual query.

The *state* of a node n is represented by the partial rewrite created by following the path from the root to n – it is obtained by concatenating the tokens on this path. The root's state is an empty rewrite, and each terminal node's state is a complete rewritten query.

Given that the vocabulary sizes of LLMs are upward of hundreds of thousands of tokens, it may become very expensive (in both financial and computational costs) to construct the entire tree. It is therefore essential to significantly reduce the token search space while exploring the tree for valid rewrites. This is precisely the purpose of MCTS which applies an *Upper Confidence Bound* (UCB) heuristic [26] to identify the best paths in a tree without computing the entire tree.

5.1 MCTS Search Process

The pseudocode of the search procedure is shown in Algorithm 2. It consists of four stages – Selection, Expansion, Simulation and Back Propagation – that are repeated across $iter_{max}$ iterations:

1. Selection: The first stage is responsible for identifying the most suitable node of the decision tree that is yet to be expanded (i.e., the tokens corresponding to this node have not yet been processed by the search procedure). Starting from the root, this process greedily selects successive edges (actions) till an unprocessed non-terminal node is reached (Lines 6–8 in Algorithm 2). Actions are picked using a UCB that balances exploration and exploitation. The goal is to pick those actions that have either (1) a higher potential to produce correct and faster rewrites (exploitation); or (2) been selected fewer times in the past (exploration).

Specifically, given a node n and a set of possible actions $a \in A$, the next node in this traversal is chosen as:

$$n_{\text{next}} = \arg\max_{a \in A} UCB(n, a) \tag{1}$$

where UCB is a heuristic sourced from [26], modified to reflect our formulation where values are associated with nodes rather than edges in the tree. It is defined as follows:

$$UCB(n,a) = V(n') + \beta(n) * P_{LLM}(a|n.state) * \frac{\sqrt{log(visits[n])}}{1 + visits[n']} \tag{2}$$

Here, n' is the node reached from n by taking action a, and the first component $V(n')$ represents the exploitation potential of n' to produce correct and faster queries (this notion is formalized below in Stage 3). The second component in the equation represents exploration – it is higher for those child nodes of n that are visited less often. Here, P_{LLM} represents the next token probability, and $visits[n]$ is the number of times n has been visited during the search process. β is a function that controls the balance between exploration and exploitation. It depends on two hyperparameters c_{base} and c – a higher value of c_{base} makes the algorithm favor exploitation , whereas a higher value of c increases the incentive to explore. β is defined as:

$$\beta(n) = log(\frac{visits[n] + c_{base} + 1}{c_{base}}) + c \tag{3}$$

2. Expansion: The second stage is used to expand the unprocessed node n_{cur} chosen by the Selection stage. Specifically, it retrieves from the LLM the top k probable next tokens from n_{cur}'s state (Line 12), and expands the decision tree by adding k new child nodes corresponding to these tokens. To make the expansion tractable, multiple child nodes are added only if the probability of the highest token falls below a threshold θ (Line 13). In other words, when the highest token probability is below θ, it means that the LLM itself is unsure of what the next token should be and therefore it is worth exploring additional options. On the other hand, if the highest token probability is greater than θ, then the tokens are

Algorithm 1 Token-augmented Rewrite

$root$ # Start State
k # Maximum number of child node expansions
θ # Probability threshold for node expansion
$iter_{max}$ # Maximum number of iterations

1: $Potential, visits, V \leftarrow$ **empty Map**
2: **for** $i \leftarrow 1, 2, \ldots, iter_{max}$ **do**
3: $\quad visits[root] \leftarrow visits[root] + 1$
4: $\quad n_{cur} \leftarrow root$
5: $\quad$ # Stage 1: Selection
6: $\quad$ **while** $len(n_{cur}.children) > 0$ **do**
7: $\quad\quad n_{cur} \leftarrow \arg\max_{a \in Actions(n_{cur}.children)} \text{UCB}(n, a)$
8: $\quad\quad visits[n_{cur}] \leftarrow visits[n_{cur}] + 1$
9: $\quad$ # Stage 2: Expansion
10: $\quad expand \leftarrow$ True
11: $\quad$ **while** $expand$ and ';' $\notin n_{cur}.state$ **do**
12: $\quad\quad tokens_{next}, P_{next} \leftarrow \text{Model}(n_{cur}, k)$
13: $\quad\quad$ **if** $P_{next}[0] \leq \theta$ **then**
14: $\quad\quad\quad$ **for** $token \in tokens_{next}$ **do**
15: $\quad\quad\quad\quad n_{new} \leftarrow$ **new** $Node$ with **State** $n_{cur}.state \cdot token$
16: $\quad\quad\quad\quad$ **Append** n_{new} to $n_{cur}.children$
17: $\quad\quad\quad expand \leftarrow$ False
18: $\quad\quad$ **else**
19: $\quad\quad\quad n_{cur}.state \leftarrow n_{cur}.state \cdot tokens_{next}[0]$
20: $\quad$ # Stage 3: Simulation - Expand from n_{cur} to full rewrite
21: $\quad query \leftarrow \text{GreedyExpand}(n_{cur})$
22: $\quad v \leftarrow \text{ComputePotential}(query)$
23: $\quad Potential[query] \leftarrow v$
24: $\quad$ # Stage 4: Backpropagation
25: $\quad$ **while** $n_{cur} \neq$ Null **do**
26: $\quad\quad V[n_{cur}] \leftarrow \max(V[n_{cur}], v)$
27: $\quad\quad n_{cur} \leftarrow \text{Parent}(n_{cur})$
28: # Return valid rewrite with maximum $Potential > 1$
29: **if** $\exists q \in Potential \mid Potential[q] > 1$ **then**
30: $\quad$ **return** q having maximum value of $Potential[q]$
31: **else**
32: $\quad$ **return** the original query

generated in a greedy fashion from the current node until a point where the LLM is again unsure of the next token, or it reaches a terminal node (i.e., completes a query rewrite).

3. Simulation: Here, we determine the potential value, $V(n_{cur})$, to be assigned to n_{cur}. This node is expanded in a greedy fashion, based on the highest-probability tokens until a terminal node is reached (Line 21). Then, the complete rewritten query represented by the terminal node's state is used to compute the potential (Lines 22–23). For a valid rewrite, $V(n_{cur})$ is equal to the *speedup* it provides with respect to the original query. However, if invalid (i.e. syntactically or semantically incorrect), $V(n_{cur})$ is assigned a zero value.

After every simulation, the complete rewritten query obtained after the greedy expansion of n_{cur} is cached along with $V(n_{cur})$ in a map, *Potential*.

4. Back Propagation: The V value of the simulation for n_{cur} is backpropagated to all its ancestor nodes. An ancestor node's V value is updated if and only if the new value is higher than the existing value.

Rewritten Query. At the end of all iterations, $q \in Potential$ with highest value $Potential[q]$ that is greater than 1 is returned as the rewritten query (Lines 29–30). In case no such rewrite exists, implying that all the valid rewrites are slower than the original query, the original query itself is returned (Line 32).

5.2 Input Prompt to MCTS

The root state in Algorithm 2 corresponds to the state just after the prompt is fed to the LLM. One way to use this algorithm is to execute it for all the various prompts discussed in the previous sections, and choose the rewrite that provides the best performance. This, however, is expensive both from the aspect of query rewrite time, as well as the number of LLM tokens used. To minimize these costs, the `LITHE` workflow first selects, given a query, the prompt yielding the most effective rewrite from among the techniques of Sections 3 and 4. It then employs this prompt to initiate the MCTS-based rewrite. In case no prompt provides a lower-cost rewrite, baseline Prompt 1 of Section 3 is used as the fallback option.

Later, in Section 7.3.3, we show how the overheads incurred can be further reduced via a classifier construction.

5.3 Performance

The performance results with MCTS-based rewrites are shown in Table 5 for the micro-benchmark. We observe gratifyingly that the SpeedupGM now attains the human speedup of **13.9**.

Table 5: Performance of MCTS-based rewrites

	# PR	SpeedupGM
`LITHE`	10	13.9

6 IMPLEMENTATION CHOICES

In this section, we briefly discuss the design choices made in our implementation of `LITHE`.

6.1 `LITHE` Parameter Settings

The *"temperature"* parameter of GPT-4o, which ranges over [0,1], controls the randomness of the model's response, with a higher temperature resulting in the outputs being more varied and less predictable. While a higher temperature can be useful for creative writing where one would seek diverse and exploratory answers, in our case we want a focused and deterministic answer as far as possible. Hence we set the temperature to 0 which causes the model to do greedy sampling to select the next token.

The hyperparameters used by `LITHE` for MCTS are as follows: The maximum number of iterations $iter_{max}$ is set to 8, expansion threshold θ is 0.7, and number of expansions k is 2. The values of c_{base} and c were set to 10 and 4, respectively. These settings were determined after an empirical evaluation of the various trade-offs, providing a robust balance between efficiency and quality.

Finally, we run the prompt pipeline for a maximum of 5 iterations to try and fix any syntax errors created during the rewrite process.

6.2 Query Efficiency Characterization

The LITHE workflow requires computing the performance of several candidate query rewrites. In this scenario, explicitly executing each rewrite on the database to evaluate its runtime benefit may become impractically slow. This is especially true for queries where even a single execution is time-consuming. We therefore leverage the *query cost*, as estimated by the optimizer, to evaluate the performance of query rewrites. It is common knowledge that these estimates are often inaccurate [16] – however, they are expected to be generally reflective of the actual run times, especially on modern industrial-strength database systems. Moreover, our comparison of costs is only in a *relative* sense – between the original query and its rewrite, so errors in the absolute values are tolerable as long as similar errors are made in all estimates. These costs can therefore serve as a computationally cheap and meaningful measure to distinguish performance among rewrites.

6.3 Query Equivalence Testing

We use a multi-stage approach, described below, to test semantic equivalence between the original query and a candidate rewrite.
1. Logic-based Equivalence. Although verifying the equivalence of a general pair of SQL queries is a long-standing hard problem [6], a variety of logic-based tools are available for proving equivalence over restricted classes of queries (e.g. Cosette[10], SQL-Solver [13], QED [30]). In our pipeline, we used the recently proposed QED [30] since it was found to cover a larger set of queries compared to the alternatives. The advantage of such a logic-based approach is that it is definitive in outcome and relatively inexpensive.
2. Result Equivalence via Sampling. If the original query is not within the QED scope, we alternatively use a sampling-based approach to test equivalence. The idea here is to execute the queries on a small sample of the database and verify equivalence based on the sample results. However, while this test is a necessary condition for query equivalence, it is not sufficient. That is, false positives may be present because the sampled database may not cover all the predicates featured in the query. To minimize this likelihood, we use a combination of (1) *correlated sampling* [34] for maintaining join integrity in the sample, (2) addition of synthetic tuples in the sample to distinguish outer and inner joins, and (3) adjusting constants in the filter predicates to produce populated results – the complete details are in the Appendix 9.
3. Result Equivalence on the Entire Database. Instead of resorting to samples, result equivalence could also be evaluated, in principle, on the entire database itself. However, as mentioned above, this could turn out to be prohibitively expensive, especially if the queries themselves are time consuming (e.g., due to the scale of the underlying database) and/or if the candidate rewrites happen to be regressions. Therefore, this option is disabled by default in our workflow, but the user can optionally invoke it as a final verification step after the sampled equivalence testing.

7 EXPERIMENTAL EVALUATION

In this section, we report on LITHE's performance profile. We first describe the experimental setup, including comparative baselines, query suites and evaluation platforms. Then we present the speedup results for both aggregate benchmark and individual queries, followed by characterization of the rewrite overheads in computational and financial terms. We finally discuss the impact of alternative platforms wrt database schema, database engine and LLMs.
Rewrite Baselines. We compare LITHE with a collection of contemporary rewrite techniques, collectively referred to as SOTA – the details of these techniques are provided in Section 8. Specifically, the SOTA collection consists of the following approaches:

1. Baseline LLM prompt [19]: This is Prompt 1 from Section 3.
2. Learned Rewrite [38], a purely rule-based rewriter.
3. LLM-R^2 [18], an LLM-guided rule-based rewriter.
4. GenRewrite [19], a purely LLM-based rewriter.

Given an input query, each of the SOTA approaches is independently invoked to perform a rewrite, and the rewrite with the best performance is used as the baseline for comparison. Note that these approaches may generate rewrites that regress the performance [19]. We discard such backsliding rewrites using, similar to LITHE, the Query-Costing module of the database engine.
Query Set. Our evaluation uses the FPR subsets from the following diverse set of benchmarks, amounting to 100+ queries:

1. TPC-DS [11]: A synthetic OLAP benchmark modeling retail supplier environment with complex analytical queries.
2. DSB [12]: A TPC-DS variant with complex data distributions and additional query templates featuring many-to-many joins and non-equi-joins.
3. ARCHER [37]: A Text-to-SQL benchmark spanning 10 databases with availability of ground-truth SQL queries.
4. JOB [16]: An optimizer stress-test benchmark featuring queries with large and complex join graphs.
5. StackOverflow [20]: A real-world benchmark with query templates modeling questions and answers from experts. A random instance of each template is taken.

Testbed. The majority of our experiments are carried out on the following data processing platform: Sandbox server with Intel(R) Xeon(R) CPU E5-1660 v4 @ 3.20GHz x 16, 32 GB RAM, and 12TB HDD, running Ubuntu 22.04 LTS; PostgreSQL v16 database engine; and GPT-4o LLM for both LITHE and SOTA. Variations on this platform are considered in Section 7.4.
Metrics. For each benchmark and each rewrite technique, we identified the number of queries for which a PR (productive rewrite) could be constructed. Subsequently, we computed the GM of the speedups obtained by each technique over the set of FPR (feasible productive rewrites). Finally, recall that if a rewrite achieves the maximum known speedup for the query, it is referred to as an MPR (Maximum Productive Rewrite).

From the investment perspective, we measured the average rewrite time per query, and for the LLM-based techniques, the number of tokens used in the rewrite process.

7.1 Rewrite Quality

7.1.1 Benchmarks. The speedup delivered by LITHE and SOTA on the FPR queries of the various benchmarks are shown in Table 6. The good news is that for TPC-DS and its DSB variant, LITHE produces PR for all the FPR queries – **65** in TPC-DS and **20** in DSB. In fact, most of these rewrites are *highly productive* as the SpeedupGM exceeds **30** for TPC-DS and **20** for DSB! Further, the PR coverage is

Table 6: Comparing LITHE with SOTA on FPR queries

Benchmark (FPR)	# PR		SpeedupGM	
	LITHE	SOTA	LITHE	SOTA
TPC-DS (65)	65	16	30.6	2.1
DSB (20)	20	3	22.6	1.3
ARCHER (25)	25	19	2.2	1.8
JOB (5)	5	2	1.8	1.3
StackOverflow (2)	2	1	9.0	7.5

far superior in comparison to SOTA– by a factor of 4 (TPC-DS) and 6 (DSB). Finally, the SpeedupGM of LITHE is higher by more than an order-of-magnitude wrt SOTA.

Turning our attention to the other benchmarks (ARCHER, JOB, StackOverflow), the number of FPR queries is smaller due to the predominance of flat SPJ formulations in these benchmarks, which limits the scope for productive rewriting. Nevertheless, LITHE continues to achieve PR in all of this limited set, whereas SOTA misses quite a few opportunities. Further, the SpeedupGM of LITHE is visibly better than SOTA.

7.1.2 Individual Queries. The above results were for entire benchmarks. We now drill down into the performance at the granularity of individual queries. Due to space limitations, we focus only on the TPC-DS benchmark here.

Figure 7 compares the performance of LITHE and SOTA on each of the 65 FPR queries in TPC-DS – note that the speedups on the x-axis are tabulated on a $\mathbf{log_{10}}$ scale. We first observe that, gratifyingly, rewrites are indeed capable of providing dramatic speedups – take, for instance, Q41, which improves by a whopping *five orders-of-magnitude* for both SOTA and LITHE. This improvement in query performance is due to replacing the "`WHERE (SELECT COUNT(*) from ...) > 0`" clause with "`WHERE EXIST (SELECT 1 from ...)`" – the latter is a more efficient check for result existence in an inner subquery since it removes the computationally expensive aggregation function.

As another example, Figure 8 shows a rewrite where only LITHE achieves 2X speedup. Here, the two subqueries in the original query (each with 3 joins) are replaced by a common set of joins, and the aggregations are conditionally computed using CASE statements.

Second, even among the 16 PR queries of SOTA, there are instances (e.g. Q6, Q47) where LITHE delivers substantially faster speedups than SOTA. Overall, across these 16 queries, LITHE attains a GM of 28.7 as compared to SOTA's GM of 19.5.

Finally, note that LITHE matched or improved on SOTA for *all* the FPR queries.

7.1.3 Query Execution Times. Thus far, we had considered optimizer estimated execution costs. We now move to wall-clock runtimes for query executions – Figure 9 shows this metric (on a $\mathbf{log_{10}}$ scale) for the original query, LITHE, and SOTA on a representative set of queries executed on the TPC-DS 100GB database.

We observe that substantial benefits are achieved, even exceeding *order-of-magnitude* improvements in some cases. Of course, as expected, these speedups may not directly match the optimizer-estimates because of the inherent imprecision in optimizer models. For example, the *5* orders-of-magnitude speedup estimated for Q41 decreases to a still respectable *3* orders-of-magnitude. The opposite

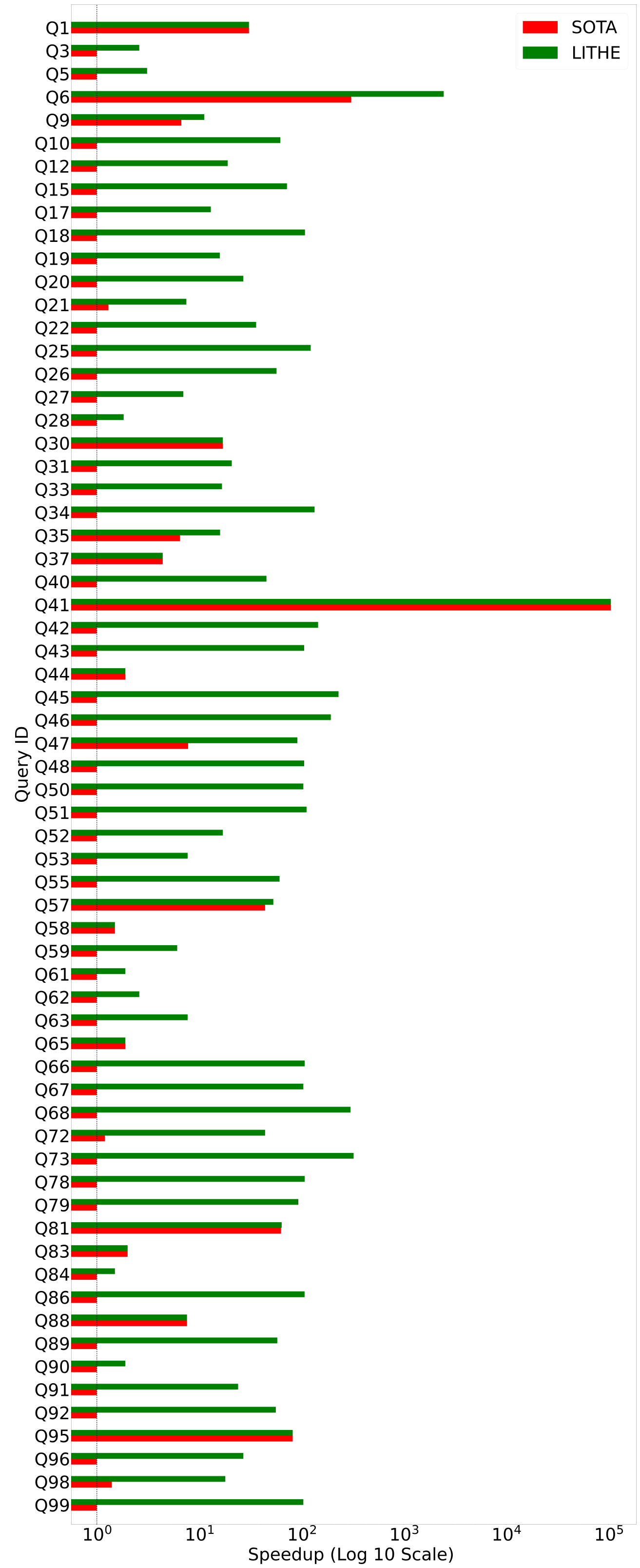

Figure 7: Speedups achieved by LITHE and SOTA

```sql
SELECT CAST(
    SUM( CASE WHEN time_dim.t_hour BETWEEN 12 AND 13
              THEN 1 ELSE 0 END) AS DECIMAL(15, 4)
  ) / CAST(
    SUM( CASE WHEN time_dim.t_hour BETWEEN 14 AND 15
              THEN 1 ELSE 0 END) AS DECIMAL(15, 4)
  ) AS am_pm_ratio
FROM web_sales
  JOIN household_demographics
    ON ws_ship_hdemo_sk = household_demographics.hd_demo_sk
  JOIN time_dim ON ws_sold_time_sk = time_dim.t_time_sk
  JOIN web_page ON ws_web_page_sk = web_page.wp_web_page_sk
WHERE household_demographics.hd_dep_count = 6
  AND web_page.wp_char_count BETWEEN 5000 AND 5200
  AND (time_dim.t_hour BETWEEN 12 AND 13
       OR time_dim.t_hour BETWEEN 14 AND 15)
ORDER BY am_pm_ratio
LIMIT 100;
```

Figure 8: Rewritten TPC-DS Q90

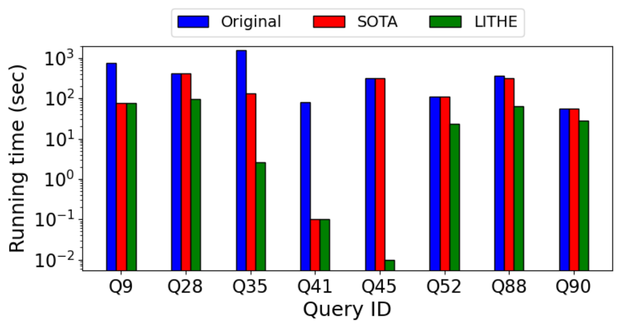

Figure 9: Query execution times on TPC-DS 100GB

also occurs: the *2* orders-of-magnitude speedup estimated for Q45 now becomes a huge *4* orders-of-magnitude.

7.2 Ablation Analysis

7.2.1 Components of LITHE. A natural question at this stage is the role of the various techniques in LITHE towards achieving its large performance benefits. This analysis is captured in Table 7 which lists the PR and MPR for each technique when invoked in *isolation*. The table also shows the *cumulative* number of PR and MPR when the different techniques are combined in the order of listing.

Table 7: Ablation Analysis

	Isolated		Cumulative	
	# PR	# MPR	# PR	# MPR
Basic Prompts (Section 3)	19	13	19	13
Rules R1 — R4 (Section 4.1)	17	12	23	15
Rules R5, R6 (Section 4.2)	44	34	60	48
MCTS (Section 5)	65	65	65	65

We observe in the table that the Basic Prompt ensemble and Redundancy Rules (R1 – R4) are capable of achieving maximum speedups for around a quarter (15/65) of the FPR query set. When the statistics-infused Rules (R5, R6) are added, this fraction jumps to about three-quarters (48/65). And finally, when MCTS is included, we reach full coverage of the FPR.

Table 8: Rewrite Overheads of LITHE and SOTA on FPR queries

Benchmark	Avg. Time (min)		Avg. Tokens		Avg. Cost USD	
	LITHE	SOTA	LITHE	SOTA	LITHE	SOTA
TPC-DS	4.6	1.7	22390	20076	0.055	0.050
DSB	8.7	4.0	22203	15699	0.055	0.039
ARCHER	2.3	0.6	7284	5465	0.018	0.013
JOB	6.1	1.3	22958	13692	0.057	0.034
StackOverflow	7.1	3.2	22634	12759	0.056	0.032

7.2.2 Database-Sensitive Prompts. Since MCTS explores the search space at a fine granularity, one could ask whether just the Basic Prompts in conjunction with MCTS would suffice to provide good performance. The motivation is that it would relieve us from using the database-sensitive rules that incur significant computational and financial overheads. When this experiment was conducted, the SpeedupGM dropped precipitously to a paltry **2.3**, a far cry from the 30.6 obtained with the database-sensitive rules. These results highlight the need to reflect database awareness for effective query rewriting, and not rely solely on prior LLM knowledge.

7.3 Rewrite Overheads

Having established the ability of LITHE to deliver performance-beneficial rewrites, we now turn our attention to the overheads incurred in this rewriting process. Here, in addition to rewrite time, the monetary outlay for LLM inferencing also has to be considered.

7.3.1 Transformation Time. The average processing time per FPR query is shown in Table 8 for LITHE and SOTA over the various benchmarks. The good news is that the rewrites are typically identified in a few minutes even for highly complex queries. We hasten to note that this investment is likely to be acceptable in deployment given that the execution benefits typically far outweigh the compilation overheads. As a case in point, for TPC-DS Q30 on a 10GB database, the original version took 17 minutes to run to completion, whereas the LITHE rewrite took 7 minutes to transform but finished in less than a second! Further, when the database size was increased to 100GB, the LITHE rewrite executed in just 8 seconds, while the original version took more than an *hour*. Finally, many applications tend to use a set of canned queries which are run thousands of times. Thus, a relatively small one-time investment can be easily recovered over repeat executions of such queries.

Notwithstanding the above, we also observe that LITHE is considerably slower than SOTA in producing rewrites. A drill-down into the various components of LITHE is shown in Table 9 for the TPC-DS queries. We observe here that the lion's-share is taken by the initial prompt ensemble and the final MCTS module, largely arising from their "brute-force" combinatorial construction. But both these modules can potentially be improved – the choice of prompt via a classifier, and MCTS by early pruning of ill-formed (semantic/syntactic errors) queries. These options are discussed below in Section 7.3.3.

7.3.2 Monetary Outlay. The average number of LLM tokens required by LITHE and SOTA for the various benchmarks, and their associated financial costs[1], are shown in Table 8. The good news is

[1]At the time of writing, GPT-4o costs USD 2.5 per million tokens.

Table 9: `LITHE` Transformation Time Analysis

	Time/Query (min)		Tokens / Query	
	Avg.	Max.	Avg.	Max.
Basic Prompts (Section 3)	1.2	5.2	2407	17778
Rules R1 — R4 (Section 4.1)	0.8	3.6	3076	11988
Rules R5, R6 (Section 4.2)	0.5	2.1	2114	5865
MCTS (Section 5)	2.1	10.6	14792	55055

that the inference charges per query are just a few cents, making rewriting practical from a deployment perspective.

We also see that `LITHE` can be a factor of two more expensive than `SOTA`. To understand the source of this expense, the token costs incurred by each module of `LITHE` are shown in Table 9. Again, the lion's share is taken by the MCTS module. This is to be expected due to its exploratory nature which can result in traversal of several paths in the tree – however, the additional expense appears to be justified by its output of significantly better rewrites. Finally, even this expense gets reduced when the time overheads are cut down, as described below.

7.3.3 Overheads Reduction. Reducing the query rewrite time while still obtaining the same performance would be possible if we could directly use the MCTS-driven rewrite with the appropriate prompt. Towards this end, we build a classifier to pick the most appropriate rewrite rule for a given input query. Specifically, the classifier identifies which, if any, of Rules R1–R6 is appropriate for a given query. If none are appropriate, then it falls back to just the set of basic prompts to identify the best prompt to be given as input to the MCTS module. In addition to reducing the rewrite times, using a classifier can also reduce the financial costs of the rewrite.

We design an LLM based classifier to accomplish this as follows. The LLM is given the rewrite rules discussed so far, and additionally, for each rule, an example demonstrating when the rule can be applied and a counter-example demonstrating when the rule cannot be applied. For the database schema and statistics-based rules, the relevant information is also fed to the classifier so that it can make an informed decision. Then, given an input query, the classifier is tasked with selecting the most appropriate rewrite rule.

Table 10 compares the performance of `LITHE` with and without the classifier. The time overheads do visibly go down by about 40 percent, and the tokens by about 20 percent. However, there is a price to be paid – the PR is reduced to 59 and the SpeedupGM comes down to 21.7. In our future work, we plan to look into whether a better tradeoff could be achieved between quality and overheads.

Pruning in MCTS. A bottleneck in the MCTS-based exploration is the need to greedily expand a node (during the simulation stage) until an entire rewrite is output. In principle, if we could quickly check for semantic and syntactic correctness at intermediate stages, then unproductive paths could be terminated early. We are currently working on the design and implementation of such checks.

7.4 Alternative Platforms (DB/Engine/LLM/T2S)

7.4.1 Anonymized Database. An interesting question to ask now is whether the performance benefits seen thus far could be an artifact of the GPT-4o LLM having already been trained well on our evaluated benchmarks, all of which are in the public domain. To investigate this issue, we created an *anonymized* version of the TPC-DS database schema, whereby the table and column names convey no semantic information about their content. We then constructed rewrites for the 65 FPR queries (after syntactic changes to reflect the new schema) on this version. The results are shown in Table 11 for `SOTA` and `LITHE`. We observe that while the SpeedupGM marginally decreases, the performance profiles remain similar to those seen earlier for native TPC-DS, testifying to the robustness of the rewrite approaches.

Table 10: Impact of Classifier (TPC-DS)

Metrics	No Classifier	With Classifier
# PR	65	59
SpeedupGM	30.6	21.7
Avg. Tokens	22390	18296
Avg. Time	4.6 min	2.8 min

Table 11: Rewrite Performance on Anonymized Database

Approach	# PR		SpeedupGM	
	TPC-DS	Masked	TPC-DS	Masked
`SOTA`	16	14	2.1	1.8
`LITHE`	65	64	30.6	29.4

7.4.2 Commercial DBMS. Another legitimate question could be whether the rewrites made amends for the relative lack of sophistication in the PostgreSQL optimizer, and that its benefits would disappear in premier database engines. To evaluate this issue, we performed TPC-DS rewrites on a popular and highly engineered commercial DBMS.

Table 12 compares the performance of `LITHE` and `SOTA` on this system. Compared to PostgreSQL, the PR reduced considerably from 65 to 26 in case of `LITHE`, while `SOTA` had just a single PR. Nevertheless, the 26 queries represent as much as a quarter of the benchmark, and their SpeedupGM is a very respectable 5.5. So, `LITHE` has a role to play even on high-end database engines. From a different perspective, a company building a new database engine could use `LITHE` to non-invasively overcome the limitations of early versions of its optimizer.

Table 12: Rewrite performance on Commercial Database

	# PR	SpeedupGM
`LITHE`	26	5.5
`SOTA`	1	1.1

7.4.3 `LITHE` on LLaMA. In our concluding experiment, we evaluate the performance on the LLaMA 3.1 70 billion parameter instruct model, substantially smaller compared to GPT-4o, which is said to have several hundred billion parameters [5]. To attain practical inference times, the model was loaded using a low 4-bit quantization. Further, to ensure reproducibility and deterministic answers, the *do_sample* parameter was set to *False*, which forces the LLM to perform greedy decoding. In order to make up for the huge reduction in model parameters as compared to GPT-4o, we include up to two example demonstrations for each rule-based prompt.

For this environment, Table 13 shows the PR and SpeedupGM obtained on the micro-benchmark with and without MCTS. Although certainly lower than the corresponding numbers with GPT-4o (see Tables 2 and 5), it is encouraging to see that, in absolute terms, significant performance benefits can be obtained for most queries, especially with MCTS support. So, the message is that even smaller models can be fruitfully used in real-world environments.

Table 13: Micro-benchmark Performance with LLaMA

	# PR	SpeedupGM
LLaMA without MCTS	5	2.6
LLaMA with MCTS	9	7.2

7.4.4 Text-to-SQL. As mention in the Introduction, we could alternatively construct the SQL-to-SQL rewrite problem as a two-stage pipeline: SQL-to-Text followed by Text-to-SQL. To assess this option, we first implemented SQL-to-Text on GPT-4o with the prompt asking for conversion of the SQL query to a clear business description. Then, two open tools were evaluated for Text-to-SQL – CHESS [28] and SQLCoder [3], which have performed very well on the BIRD [17] and SQL-Eval [2] benchmarks, respectively. However, in all our experiments, the outcomes were uniformly poor with the rewrites often suffering semantic errors, and even among the few correct rewrites, there was not a single PR. Therefore, this alternative architecture does not appear to be a viable option.

8 RELATED WORK

Rule-based SQL rewriting. Most of the recent work on SQL query rewriting is rule-based [7, 9, 22, 31, 32, 38]. For instance, WeTune [31] employs a rule generator to enumerate a set (up to a maximum size) of logically valid plans for a given query to create new rewrite rules, and uses an SMT solver to prove the correctness of the generated rules. While this approach can generate a large set of new rewrite rules, it often fails in coming up with transformation rules for complex queries due to the computational overheads of verifying rule correctness. As such, it is unable to rewrite any of the TPC-DS queries [14, 19].

Learned Rewrite [38] uses a set of existing rewrite rules from Calcite [8] and aims to learn the optimal subset of rules along with the order in which they must be applied. Given that the search space for possible rewrites is exponential in the number of applicable rules, an MCTS scheme is used to efficiently navigate this space and find the rewritten query with maximum cost reduction.

LLM-R^2 [18] is also rule-based but takes a different approach to identify the order for rewrite rule applications: it uses an LLM to find the best Calcite rules and the order in which to apply them to improve the query performance. R-Bot [27] also leverages an LLM to optimize the order of Calcite rules, but employs advanced contemporary techniques such as retrieval-augmented generation (RAG) and step-by-step self-reflection to improve the outcomes.

Query Booster [7] implements human centred rewriting – it provides an interface that the user can use to specify rules using an expressive rule language, which it generalizes to create rewrite rules to be applied on the query. There also exist other rule-based rewrite approaches that are designed for specific type of rewrites such as optimizing correlated window aggregations [32] and common expression elimination [9].

All of the above approaches operate via the query *plan space*, which can restrict the kind of rewrites that can be accomplished. Whereas, `LITHE` uses a small set of general rewrite rules that work directly in the *query space*.

LLM-based rewriting. GenRewrite [19] is the first LLM-based approach that tries to use the LLM itself for end-to-end query rewriting. Instead of using predefined rules from Calcite [8], they employ the LLM to create Natural Language Rewrite Rules (NLR2s) to be used as hints, and perform several iterations of prompting to get the rewritten query. They show that LLMs can outperform rule-based approaches due to their ability to understand contexts, and demonstrate a significant improvement in query rewriting compared to prior methods.

A limitation, however, is that LLM-generated rewrite rules often fail to generalize beyond specific query pairs. Even when generalized rules are present, without accompanying examples, LLMs can struggle to apply rules correctly. Finally, it must be noted that LLMs are unaware of the underlying database which restricts their ability to produce efficient metadata-aware rules.

LLMs for Text-to-SQL. LLMs have also been extensively used for Text-to-SQL transformations [2, 3, 17, 28, 33]. The main focus of these techniques is to correctly ascertain the information necessary to formulate the SQL query [23, 28, 29]. On the other hand, the goal of SQL rewriting is on improving the performance of an existing SQL query. Thus, unlike Text-to-SQL transformations where the input text is inherently ambiguous, SQL queries are precisely defined, and therefore equivalence to a precise ground-truth has to be provably maintained.

9 CONCLUSIONS AND FUTURE WORK

We investigated how the latent power of LLM technologies can be productively materialized in the context of SQL-to-SQL rewriting. Our study progressively infused database domain knowledge, such as redundancy removal rules and schematic+statistical metadata, into the LLM prompts. Further, the output telemetry of LLMs, in the form of token probabilities, was used to signal situations where the LLM lacked confidence, triggering exploration of a larger search space. Finally, a combination of logic-based and statistical tests was employed to verify the equivalence of the rewrites. An empirical evaluation over common database benchmarks showed that rewriting is a potent mechanism to improve query performance.

The speedups obtained are significant and proof-of-concept, but there remains scope for improvement. For instance, while a speedup of 83X was produced for Q95 of TPC-DS, we were able to hand-craft an alternate rewrite with 99X speedup by moving a self-join from a CTE to the main query, thereby reducing scans of the WEB_SALES table. We are currently working on adding a few select rules to implement such optimizations within the `LITHE` framework.

Our focus here was primarily on prompting-based strategies. In our future research, we also plan to investigate how domain-specific *fine-tuning* could be leveraged to provide GPT-4o-like rewrites on small open models.

REFERENCES

[1] 2023. PostgreSQL Release 16. www.postgresql.org/docs/16/release-16.html

[2] 2024. Defog SQL-Eval. https://github.com/defog-ai/sql-eval

[3] 2024. Defog SQLCoder. github.com/defog-ai/sqlcoder

[4] 2024. GroupBy with joined table produces sub optimal query. github.com/npgsql/efcore.pg/issues/3160

[5] 2024. OpenAI's GPT 4o vs GPT 4o mini: Which AI model to use and why. https://timesofindia.indiatimes.com/technology/tech-tips/openais-gpt-4o-vs-gpt-4o-mini-which-ai-model-to-use-and-why/articleshow/111927368.cms

[6] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Equivalences Among Relational Expressions. *SIAM J. Comput.* 8, 2 (1979), 218–246. https://doi.org/10.1137/0208017

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

[8] 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 *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 221–230. https://doi.org/10.1145/3183713.3190662

[9] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. 2022. Computation reuse via fusion in Amazon Athena. In *ICDE 2022*. www.amazon.science/publications/computation-reuse-via-fusion-in-amazon-athena

[10] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Conference on Innovative Data Systems Research*. api.semanticscholar.org/CorpusID:12408033

[11] The Transaction Processing Performance Council. 2006. TPC Benchmark™ DS (TPC-DS). In *TPC Benchmark DS (Decision Support)*. www.tpc.org/tpcds/

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

[13] 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, Article 227 (dec 2023), 26 pages. https://doi.org/10.1145/3626768

[14] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. 2023. SlabCity: Whole-Query Optimization Using Program Synthesis. *Proceedings of the VLDB Endowment* 16 (08 2023), 3151–3164. https://doi.org/10.14778/3611479.3611515

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endow.* 9, 3 (Nov. 2015), 204–215. https://doi.org/10.14778/2850583.2850594

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

[17] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Ma Chenhao, Guoliang Li, Kevin Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 42330–42357. proceedings.neurips.cc/paper_files/paper/2023/file/83fc8fab1710363050bbd1d4b8cc0021-Paper-Datasets_and_Benchmarks.pdf

[18] 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. arXiv:2404.12872 [cs.DB] arxiv.org/abs/2404.12872

[19] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. arXiv:2403.09060 [cs.DB] arxiv.org/abs/2403.09060

[20] Ryan Marcus. 2021. Stack dataset. rmarcus.info/stack.html

[21] Microsoft. [n.d.]. Entity Framework Documentation Hub. learn.microsoft.com/en-us/ef/

[22] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *SIGMOD Rec.* 21, 2 (June 1992), 39–48. https://doi.org/10.1145/141484.130294

[23] Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: Decomposed In-Context Learning of Text-to-SQL with Self-Correction. In *Thirty-seventh Conference on Neural Information Processing Systems*. openreview.net/forum?id=p53QDxSIc5

[24] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., USA.

[25] Morgane Rouvroy. 2022. Join Elimination. yellowbrick.com/blog/yellowbrick-engineering/join-elimination/

[26] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. 2017. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. arXiv:1712.01815 [cs.AI] arxiv.org/abs/1712.01815

[27] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2024. R-Bot: An LLM-based Query Rewrite System. arXiv:2412.01661 [cs.DB] https://arxiv.org/abs/2412.01661

[28] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. 2024. CHESS: Contextual Harnessing for Efficient SQL Synthesis. https://doi.org/2405.16755

[29] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, and Zhoujun Li. 2024. MAC-SQL: A Multi-Agent Collaborative Framework for Text-to-SQL. arXiv:2312.11242 [cs.CL] arxiv.org/abs/2312.11242

[30] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (aug 2024), 3602–3614. https://doi.org/10.14778/3681954.3682024

[31] 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 *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. https://doi.org/10.1145/3514221.3526125

[32] Wentao Wu, Philip A. Bernstein, Alex Raizman, and Christina Pavlopoulou. 2020. Factor Windows: Cost-based Query Rewriting for Optimizing Correlated Window Aggregates. *2022 IEEE 38th International Conference on Data Engineering (ICDE)* (2020), 2722–2734. api.semanticscholar.org/CorpusID:244117293

[33] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Danrui Qi, Hong Yi, Shaodong Liu, and Faqiang Chen. 2024. DB-GPT: Empowering Database Interactions with Private Large Language Models. arXiv:2312.17449 [cs.DB] arxiv.org/abs/2312.17449

[34] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. 2013. CS2: a new database synopsis for query estimation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 469–480. https://doi.org/10.1145/2463676.2463701

[35] Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, Zilin Zhang, and Dragomir Radev. 2018. Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Ellen Riloff, David Chiang, Julia Hockenmaier, and Jun'ichi Tsujii (Eds.). Association for Computational Linguistics, Brussels, Belgium, 3911–3921. https://doi.org/10.18653/v1/D18-1425

[36] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. 2023. Planning with Large Language Models for Code Generation. In *The Eleventh International Conference on Learning Representations*. openreview.net/forum?id=Lr8cOOtYbfL

[37] Danna Zheng, Mirella Lapata, and Jeff Z. Pan. 2024. Archer: A Human-Labeled Text-to-SQL Dataset with Arithmetic, Commonsense and Hypothetical Reasoning. In *Conference of the European Chapter of the Association for Computational Linguistics*. api.semanticscholar.org/CorpusID:267760297

[38] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. 2023. A Learned Query Rewrite System. *Proc. VLDB Endow.* 16, 12 (aug 2023), 4110–4113. https://doi.org/10.14778/3611540.3611633

APPENDIX

A1 : Detailed MCTS Algorithm

Algorithm 2 Token-augmented Rewrite

$root$, # Start State;
k, # Maximum number of child expansion
θ, # Probability threshold for node expansion
$iter_{max}$, # Maximum number of iterations
$Potential, visits, V \leftarrow$ **empty Map**
$i \leftarrow 1, 2, \ldots, iter_{max}$
$visits[root] \leftarrow visits[root] + 1$
$n_{cur} \leftarrow root$
\# Stage 1: Selection $len(n_{cur}.children) > 0$
$n_{cur} \leftarrow \arg\max_{a \in Actions(n_{cur}.children)}$ UCB(n, a)
$visits[n_{cur}] \leftarrow visits[n_{cur}] + 1$
\# Stage 2: Expansion
$expand \leftarrow$ True $expand$
$tokens_{next}, P_{next} \leftarrow$ Model(n_{cur}, k) $tokens_{next}[0] =$‘;’
$expand \leftarrow$ False $P_{next} \leq \theta$ $token \in tokens_{next}$
$state \leftarrow n_{cur}.state \cdot token$
$n_{new} \leftarrow$ **new** $Node$ with **State** $state$
Append n_{new} to $n_{cur}.children$
$expand \leftarrow$ False
$state \leftarrow n_{cur}.state \cdot tokens_{next}[0]$ # First Token
$n_{new} \leftarrow$ **new** $Node$ with **State** $state$
Append n_{new} to $n_{cur}.children$
$n_{cur} \leftarrow n_{new}$
\# Stage 3: Simulation
$query \leftarrow$ GreedyExpand(n_{cur})
\# Compute the potential of $query$ (returns 0 for invalid $query$)
$v \leftarrow$ ComputePotential$(query)$
$Potential[query] \leftarrow v$
\# Stage 4: Backpropagation
$n \leftarrow n_{cur}$ $n \neq$ Null $v > V[n]$
$V[n] \leftarrow v$
$n \leftarrow$ Parent(n)
\# Return valid rewrite with maximum $Potential > 1$ $\exists q \in Potential \mid Potential[q] > 1$
return q having maximum value of $Potential[q]$
return the original query

A2 : Query Equivalence Testing

We use a multi-stage approach, described below, to test semantic equivalence between the original query and a candidate rewrite.

1. Logic-based Equivalence. Although verifying the equivalence of a general pair of SQL queries is a long-standing hard problem [6], a variety of logic-based tools are available for proving equivalence over restricted classes of queries (e.g. Cosette[10], SQL-Solver [13], QED [30]). In our pipeline, we used the recently proposed QED [30] since it was found to cover a larger set of queries compared to the alternatives. The advantage of such a logic-based approach is that it is definitive in outcome and relatively inexpensive.

2. Result Equivalence via Sampling. If the original query is not within the QED scope, we alternatively use a sampling-based approach to test equivalence. The idea here is to execute the queries on a small sample of the database and verify equivalence based on the sample results. However, while this test is a necessary condition for query equivalence, it is not a sufficient condition. That is, there are no false negatives, but there can be false positives. This is because the sampled database may not cover all the predicates present in the query. This can cause two types of problems: First, it is possible for two different queries to return the same result. This can happen when, for example, the entirety of the sampled data satisfies a predicate of one query, while the same predicate is not present in the other. Second, if the underlying sample does not satisfy any of the predicates in either query, then an empty result will be returned by both queries. This again does not imply that the queries are equivalent.

To probabilistically address the first problem (false positives), we create multiple samples of the database with different seeds, and run the test on all these samples. The goal is to reduce the likelihood of non-equivalent queries returning the same results.

To minimize the occurrence of the second problem (empty results), the following approach is taken:

(1) We use *correlated sampling* [34] to sample the database. This technique leverages the join graph of the schema to produce a sample that maintains join integrity between the tables participating in the query.
(2) To differentiate between inner and outer joins, we insert into the sample database, rows with appropriate values on the relevant columns such that they produce NULL values for outer joins.
(3) Given a pair of queries to test for equivalence, we adjust the constants in the filter predicates to reduce the chances of an empty result. We make use of the rows in the sampled data for this purpose. For example, say an equality predicate is present in the query and the associated constant is absent in the sampled database. We then replace the query constant with a value already present in the sample. Similarly, the constants for other comparison operators are adjusted based on the ranges of the corresponding columns in the sampled database. Note that these modifications only change the selectivity of the query, but not its semantics.

3. Result Equivalence on the Entire Database. Instead of resorting to samples, result equivalence could also be evaluated, in principle, on the entire database itself. However, this could turn out to be prohibitively expensive, especially if the queries themselves are time consuming (e.g., due to the scale of the underlying database) and/or if the candidate rewrites happen to be regressions. Therefore, this option is disabled by default in our workflow, but the user can optionally invoke it as a final verification step after the sampled equivalence testing.

A3 : Examples used in Rule1-Rule6

R1: Use CTEs (Common Table Expressions) to avoid repeated computation.

Original Query

```
SELECT emp.employee_name,
       mgr.manager_name
FROM   employees emp,
       managers mgr
WHERE  emp.manager_id = mgr.manager_id
       AND emp.employee_id IN (SELECT manager_id
                               FROM   (SELECT manager_id,
                                              manager_name
                                       FROM   managers
                                       WHERE  job_id = 'IT_PROG'
                                              AND manager_id > 100))
       AND mgr.manager_name IN (SELECT manager_name
                                FROM   (SELECT manager_id,
                                               manager_name
                                        FROM   managers
                                        WHERE  job_id = 'IT_PROG'
                                               AND manager_id > 100));
```

Rewritten Query

```
WITH cte
     AS (SELECT manager_id,
                manager_name
         FROM   managers
         WHERE  job_id = 'IT_PROG'
                AND manager_id > 100)
SELECT emp.employee_name,
       mgr.manager_name
FROM   employees emp,
       managers mgr
WHERE  emp.manager_id = mgr.manager_id
       AND emp.employee_id IN (SELECT manager_id
                               FROM   it_prog_managers)
       AND mgr.manager_name IN (SELECT manager_name
                                FROM   it_prog_managers);
```

R2: When multiple subqueries use the same base table, rewrite to scan the base table only once.

Original Query

```
SELECT (SELECT Avg(salary)
        FROM   employees
        WHERE  department = 'Sales'
               AND experience_years BETWEEN 1 AND 5
               AND salary BETWEEN 50000 AND 60000) AS Sales_Avg,
       (SELECT Avg(salary)
        FROM   employees
        WHERE  department = 'HR'
               AND experience_years BETWEEN 5 AND 10
               AND salary BETWEEN 80000 AND 90000) AS HR_Avg;
```

Rewritten Query

```
SELECT avg(
       CASE
              WHEN department = 'Sales' THEN salary) AS sales_avg,
       avg(
       CASE
              WHEN department = 'HR' THEN salary) AS hr_avg
FROM   employees
WHERE  (
              department = 'Sales'
       AND    experience_years BETWEEN 1 AND    5
       AND    salary BETWEEN 50000 AND    60000)
OR     (
              department = 'HR'
       AND    experience_years BETWEEN 5 AND    10
       AND    salary BETWEEN 80000 AND    90000);
```

R3: Eliminate overlapping subqueries.

Original Query

```
SELECT c.*
FROM   customer c
WHERE  c.address_id IN (SELECT a.address_id
                        FROM   address)
       AND c.address_id IN (SELECT a.address_id
                            FROM   address
                            WHERE  a.pin_code = '560012');
```

Rewritten Query

```
SELECT c.*
FROM   customer c
WHERE  c.address_id IN (SELECT a.address_id
                        FROM   address
                        WHERE  a.pin_code = '560012');
```

R4: Remove unnecessary joins between a primary key and a foreign key.

Schema

```
CREATE TABLE products
  (
     p_product_id INTEGER NOT NULL,
     PRIMARY KEY (p_product_id)
  );

CREATE TABLE fact_sales
  (
     f_sales_id   INTEGER NOT NULL,
     f_units_sold INTEGER NOT NULL,
     f_product_id INTEGER NOT NULL,
     PRIMARY KEY (f_sales_id),
     FOREIGN KEY (f_product_id) REFERENCES products(p_product_id)
  );
```

Original Query

```
SELECT p_product_id,
       f_units_sold
FROM   fact_sales,
       products
WHERE  f_product_id = p_product_id;
```

Rewritten Query

```
SELECT f_product_id,
       f_units_sold
FROM   fact_sales;
```

R5: Choose EXIST or IN based on subquery selectivity.

Original Query

```
Select item.id, item.code, item.price
from item
where item.sourceid in (
    Select element.sourceid
    from element
    where element.zip > 1100
  )
order by item.id;
```

Statistics

```
Selectivity of different predicates is given below :
( 1 ) source_id > 1100 on table element :: 0.7385
```

Rewritten Query

```
Select item.id, item.code, item.price
from item
where exists(select 1
    from element
    where item.sourceid = element.sourceid
      and element.sourceid > 1100
  )
order by item.id;
```

R6: Pre-filter fact tables in a CTE using those dimension tables having low selectivity values. Retain dimension table filters in main query. Do not create explicit join statements.

Original Query

```
Select item.id, item.code, item.price, country.name
from item, element, country, colour
where element.sourceid = item.sourceid
  and item.code = country.code
  and item.colorcode = colour.colorcode
  and element.zip > 8000
  and country.name = 'US'
  and colour.colour_name = 'Red'
order by item.id;
```

Statistics

```
Selectivity of different predicates is given below :
( 1 ) element.zip > 8000 :: 0.002
( 2 ) country.name = 'US' :: 0.4
( 3 ) colour.colour_name = 'Red' :: 0.01
```

Rewritten Query

```
with filtered_item as(
  Select *
  from item
  where
    item.sourceid in (
      Select
        element.sourceid
      where
        element.zip > 8000
    )
    and item.colorcode in (
      Select
        colour.colorcode
      from
        colour
      where
        colour.colour_name = 'Red'
    )
)
Select item.id, item.code, item.price, country.name
from filtered_item as item, element, country, colour
where
  item.sourceid = element.sourceid
  and item.code = country.code
  and item.colorcode = colour.colorcode
  and element.zip > 8000
  and country.name = 'US'
  and colour.colour_name = 'Red'
order by item.id;
```