Robust Query Transformation using LLMs (Machine Learning Aspects)

A PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

Master of Technology

IN

Faculty of Engineering

BY

Sriram Dharwada

Computer Science and Automation
Indian Institute of Science
Bangalore – 560 012 (INDIA)

June, 2025

Declaration of Originality

I, **Sriram Dharwada**, with SR No. **04-04-00-10-51-23-1-23096** hereby declare that the material presented in the thesis titled

Robust Query Transformation using LLMs (Machine Learning Aspects)

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **Years**.
With my signature, I certify that:

- I have not manipulated any of the data or results.
- I have not committed any plagiarism of intellectual property. I have clearly indicated and referenced the contributions of others.
- I have explicitly acknowledged all collaborative research and discussions.
- I have understood that any false claim will result in severe disciplinary action.
- I have understood that the work may be screened for any form of academic misconduct.

Date: 23/06/2025 Student Signature

In my capacity as supervisor of the above-mentioned work, I certify that the above statements are true to the best of my knowledge, and I have carried out due diligence to ensure the originality of the report.

Advisor Name: Prof. Jayant Haritsa Advisor Signature

© Sriram Dharwada
June, 2025
All rights reserved

DEDICATED TO

My Parents

for their constant love and support

Acknowledgements

First and foremost, I owe my deepest gratitude to Prof. Jayant Haritsa, my project advisor, whose guidance has been the compass steering this work. His sharp insights, never-give-up attitude, and constant encouragement were transformative in shaping me. I am equally thankful to Dr. Harish Doraiswamy, Principal Researcher at Microsoft Research India, for opening the doors to this project through an internship opportunity at Microsoft. It has been a privilege and an invaluable learning experience to work under the mentorship of both of you at IISc.

To my brilliant project partner, Himanshu Devrani, I extend heartfelt thanks. Your steady presence through both the smooth stretches and the challenging phases of this journey meant more than words can say. I'm deeply grateful for your support, collaboration, and friendship throughout this project.

I am also grateful to the faculty in IISc for instilling in me a spark of learning, and to my friends from CSA and DSL who have made my stay at IISc memorable.

Above all, I am deeply indebted to my family. Your steadfast support and belief in me have been the backbone of this journey. I'm deeply grateful for your love and encouragement every step of the way.

Abstract

Query rewriting is a classical technique for transforming complex declarative SQL queries into simpler equivalents, improving execution speed and developer comprehension. This is typically done using transformation rules, which are limited in scope and hard to update in production. Emerging methods using large language models (LLMs) show promise but often suffer from semantic errors and syntactic inconsistencies.

This thesis shows how LLMs' reasoning abilities can be used for reliable query optimization, employing generic and database-aware prompts, LLM token probability-guided rewrites, and a self-reflective LLM agent. To ensure correctness and stability, we incorporate a wide range of statistical and logic-based validation mechanisms. We implemented these LLM-driven techniques in the `LITHE` and `AGENTIC-LITHE` systems and evaluated them on complex analytic queries from standard benchmarks on modern database platforms.

Tests on industry-standard benchmarks show our system outperforms state-of-the-art techniques by an order of magnitude, serving as a reliable intermediary between enterprise applications and databases. For example, on TPC-DS with PostgreSQL, slow queries saw a geometric mean speedup of **10.2**× over the native optimizer, compared to **4.9**× from SOTA. Overall, `LITHE` and `AGENTIC-LITHE` mark a promising step toward practical LLM-based advisory tools for enterprise query performance.

This investigation is a joint project with another M.Tech CSA student, Himanshu Devrani. My thesis focuses on detailing the components and work implemented by me, while the remaining aspects of the project are covered in the technical report [10].

Publications based on this Thesis

Sriram Dharwada, Himanshu Devrani, Jayant R. Haritsa, and Harish Doraiswamy. Query rewriting via llms. CoRR, abs/2502.12918, 2025. doi: 10.48550/ARXIV.2502.12918. URL https://doi.org/10.48550/arXiv.2502.12918.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SQL queries in enterprise applications are often burdened with inefficiencies, particularly when generated by tools like ORM frameworks. A clear illustration of this issue is the blog-processing query presented in Figure 1.1, which was created using the widely used Entity Framework [20]. This complex query, intended to generate a daily summary of rating metrics for highly-rated blogs, can be simplified into a more efficient flat query, as demonstrated in Figure 1.2.

Simplifying complex query structures into lean equivalents offers numerous advantages. First, it significantly enhances query readability, making it easier to debug and maintain queries in industrial settings. Second, while query optimizers are theoretically capable of eliminating redundancies to create efficient execution plans, in practice, they often struggle with overly complex query structures, leading to suboptimal performance. In fact, one of the most popular database optimizers, PostgreSQL [1], failed to optimize the query shown in Figure 1.1.

This issue arises because the optimizer typically performs optimizations at the node level in the execution plan, where it lacks the context to fully understand the declarative meaning of the query. As a result, it cannot perform meaningful transformations at the query level. In contrast, large language models (LLMs), with their advanced context understanding, can interpret the query semantically and transform it into more efficient SQL instructions. Therefore, an LLM based query re-writer can serve as an effective and non-invasive mechanism for delivering good performance despite inherent optimizer limitations. The non-invasive nature of LLM-based query rewriting can enable open-source optimizers to match the performance of commercial database optimizers, bypassing the complexity of modifying the optimizer itself.

```
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 1.1: Complex SQL Representation

```
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 1.2: Lean Equivalent Query

Building on this context, we have designed an effective SQL-to-SQL query transformation tool which meets the following essential criteria: (1) The rewritten query must preserve the semantic equivalence with the original query; (2) The transformation should avoid any degradation in execution efficiency; and (3) The computational and resource costs of performing the transformation should remain feasible for real-world deployment scenarios.

This investigation is a joint project with another M.Tech CSA student, Himanshu Devrani. My thesis focuses on detailing the components and work implemented by me, while the remain-

ing aspects of the project are covered in the technical report [10].

1.1 Query Rewriting via LLMs

1.1.1 Prior Work

A significant portion of modern SQL query rewriting research focuses on rule-based methodologies [38, 31, 5, 33, 7, 22]. For example, WeTune [31] generates new rewrite rules by enumerating a bounded set of logically equivalent query plans and employing an SMT solver to verify their correctness. Despite its ability to produce numerous rewrite rules, this approach struggles to handle complex queries due to the computational challenges associated with rule validation. As such it is unable to optimize any of the TPC-DS queries [13, 17].

Learned Rewrite [38], by contrast, builds on Calcite's [6] predefined set of rewrite rules. It uses Monte Carlo Tree Search (MCTS) to navigate the combinatorial explosion of possible rule sequences, aiming to identify the most effective subset and application order for these rules. Similarly, LLM-R^2 [16] also employs Calcite's rules but utilizes an LLM to determine the optimal rules and their sequence for improving query performance. R-Bot [27] extends this concept by integrating advanced methods like retrieval-augmented generation (RAG) and iterative self-reflection to further enhance the ordering of Calcite rules. Query Booster [5], on the other hand, enables users to define custom rewrite rules through a flexible rule language. These user-defined rules are then generalized for application to SQL queries.

All of the above approaches operate via the query *plan space*, (i.e optimization on the nodes of the execution plan tree) rather than directly in *query space* (i.e transforming the query structure itself), which can restrict the kind of rewrites that can be accomplished.

GenRewrite [17] is the first LLM-based approach that tries to use the LLM itself for end-to-end query rewriting. Rather than relying on Calcite's [6] predefined rules, they use the LLM to generate Natural Language Rewrite Rules (NLR2s), which act as hints for query rewriting. Through iterative prompting, it refines these rules to produce rewritten queries. The results highlight the ability of LLMs to outperform rule-based methods by better understanding query contexts, resulting in notable advancements in query rewriting compared to previous techniques.

However, LLM-generated rewrite rules often fail to generalize beyond specific queries, and applying them correctly can be challenging without clear examples. Furthermore, LLMs' lack of database awareness limits their ability to create efficient, metadata-informed rewrites.

1.1.2 The LITHE Rewriter

To address these limitations, we present LITHE (LLM Infused Transformations of HEfty queries), an LLM-based query rewriting assistant to aid DBAs in tuning slow-running queries that have entailed their intervention. The system is built through a structured exploration of prompt engineering strategies. It features a suite of basic prompts as well as database-sensitive prompts with Redundancy Removal and Metadata-infused Rules.

In addition to prompt-based strategies, LITHE exploits token probability outputs from the LLM to identify uncertain predictions. When the model is unsure about token generation, it branches into multiple alternative paths using Monte Carlo Tree Search (MCTS) to explore potentially better rewrites.

To ensure rewritten queries are both correct and performant, LITHE verifies semantic equivalence using a combination of statistical result comparisons and logic-based tools. Additionally, it employs heuristics to detect and discard rewrites that may regress in actual runtime performance, despite favorable optimizer estimates.

1.1.3 Results

Our first set of experiments to evaluate LITHE's performance is carried out on the industry standard TPC-DS benchmark [8], hosted on the PostgreSQL platform with GPT-4o used as the LLM. The evaluation focuses on slow queries taking more than a threshold time to complete. We compare the performance of LITHE against SOTA techniques (specifically, Learned Rewrite [38], LLM-R^2 [16], GenRewrite [17], as well as a baseline LLM prompt [17]). The primary metrics are (a) reductions in optimizer-estimated costs, (b) run-time speedups, and (c) rewriting overheads. For LLM-based techniques, the number of tokens used is also monitored since the financial charges for LLM usage are typically dependent on this number. To understand the independent utility of the various components of LITHE, a systematic ablation study is carried out.

In our second stage of experiments, we evaluate generalizability of the above outcomes in a variety of new scenarios, including (a) unseen database schemas, and (b) alternative LLM platforms.

Our experiments demonstrate that LITHE achieves, for many slow queries, semantically correct transformations that significantly reduce the abstract costs. In particular, for TPC-DS, LITHE constructed "highly productive" ($> 1.5x$ estimated speedup) rewrites for as many as 26 queries, whereas SOTA promised such rewrites for only about half the number. Further, the GM (Geometric Mean) of LITHE's cost reductions reached **11.5**, almost double the **6.1** offered by SOTA.

We also evaluated whether the above cost reductions translated into real execution speedups. Here, we find that LITHE is indeed often substantively faster at run-time as well. Specifically, the geometric mean of the runtime speedups for slow queries was as high as **13.2** over the native optimizer, whereas SOTA delivered **4.9** in comparison.

Overall, LITHE is a promising step toward viable LLM-based advisory tools for ameliorating enterprise application performance.

1.2 Agentic AI for Query Rewriting

1.2.1 Prior Work

In recent times, **Agentic reasoning** has been incorporated into LLMs, enabling them to dynamically interact with external environments and to reflect on their past steps to improve the quality of chosen actions [26, 21]. In particular, they have been extensively used for Text-to-SQL transformations [29, 34, 32, 24, 28]. The main focus of these Agentic techniques is to correctly ascertain the information necessary to formulate the SQL query [29, 23, 28]. On the other hand, the goal of S2S rewriting is on improving the performance of an existing SQL query.

1.2.2 The AGENTIC-LITHE Rewriter

Building on LITHE, we propose an enhanced framework, AGENTIC-LITHE, that introduces a reasoning-driven LLM Agent for S2S transformations. Inspired by the Tree of Thoughts problem-solving method [35], this Agent can self-analyze its errors and devise new SQL-to-SQL rewrite rules that go beyond those used in LITHE. Rather than generating a complete SQL rewrite in a single step, the Agent first reflects on potential optimizations, then applies them incrementally, improving rewrite quality by fostering deeper reasoning and adaptive correction.

1.2.3 Results

On the same test bed used to evaluate LITHE, AGENTIC-LITHE demonstrated superior performance compared to both LITHE as well as SOTA. For TPC-DS, AGENTIC-LITHE constructed 4 additional highly productive rewrites for queries that neither LITHE nor SOTA were able to optimize. Taking these additional rewrites into account, the GM of AGENTIC-LITHE's cost reductions reached **10.2**, whereas those of LITHE and SOTA fell to **8.4** and **4.9**, respectively. Similarly, for slow queries, its GM runtime speedup reached **17.7**, significantly higher than LITHE (**9.4**) and the state-of-the-art (**4.0**).

Next, AGENTIC-LITHE was tested for generalizability on an unseen database schema as well as an alternative LLM platform. In both cases, it continued to outperform LITHE and SOTA.

An ablation study was also carried out to assess the contributions and rewrite overheads of the different components of `AGENTIC-LITHE` in isolation.

Finally, an in-depth analysis of the Agent's reasoning demonstrated that `AGENTIC-LITHE` could independently discover rewrite rules employed by `LITHE`, and beyond.

1.3 Contributions

In summary, our study makes the following contributions:

1. Assesses LLM suitability for S2S transformation.
2. Transforms directly in query space instead of plan space intermediates, leading to performant rewrites.
3. Leverages LLM token probabilities to guide navigation of the rewrite search space and minimize LLM errors.
4. Uses an LLM agent to iteratively analyze, refine, and generate optimized SQL rewrites.
5. Evaluates rewrite quality over a broad range of database environments, demonstrating substantial benefits over both SOTA and the native optimizer.
6. Identifies learnings that could help guide research directions for industrial-strength query rewriting.

1.4 Overview

The remainder of this thesis is structured as follows: Chapter 2 describes the architecture and overall design of `LITHE`. Chapter 3 explores the Token Probability Driven Rewrite pipeline employed in `LITHE`. Chapter 4 presents a comprehensive experimental evaluation of `LITHE`, while Chapter 5 discusses its limitations. Chapter 6 introduces `AGENTIC-LITHE`, a novel LLM Agent based S2S transformation tool, with its evaluation covered in Chapter 7. Finally, Chapter 8 outlines future research directions, and Chapter 9 concludes the thesis.

Chapter 2

LITHE Architecture

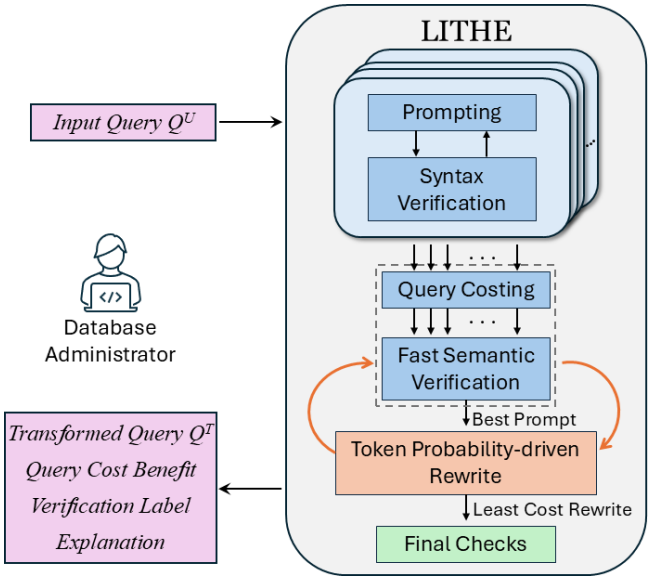

Figure 2.1: High-level architecture of LITHE

In the initial phase of this study, we conducted a calibrated investigation into the reliable use of LLMs for rewriting complex SQL queries. This work led to the development of the LITHE (LLM Infused Transformations of Hefty queries) system. As illustrated in the architectural diagram of Figure 2.1, LITHE takes as input the user query Q^U and outputs a transformed query Q^T, together with (a) the *expected* performance improvement, in terms of optimizer estimated cost, of Q^T; (b) a *verification label* indicating the mechanism (provable or statistical) used to determine that Q^T is semantically equivalent to Q^U; and (c) a *reasoning* for why the LLM expects Q^T to be helpful wrt performance. This information allows DBAs to make informed decisions about adopting Q^T. Notably, keeping a DBA in the loop is standard practice

in commercial query advisory systems [9].

Our design of LITHE was based on a calibrated investigation of the suite of techniques described below:

Prompt-based rewriting pipeline.

The pipeline feeds the user query and an ensemble of prompts to the LLM prompting module in parallel, each requesting a rewrite. The prompts range from a set of Basic Prompts, to database sensitive prompts that incorporate rewrite rules commonly followed by DBAs. The Basic Prompts, shown in Figure 2.2, cover a progressive range of instruction detail to test the LLM's base knowledge.

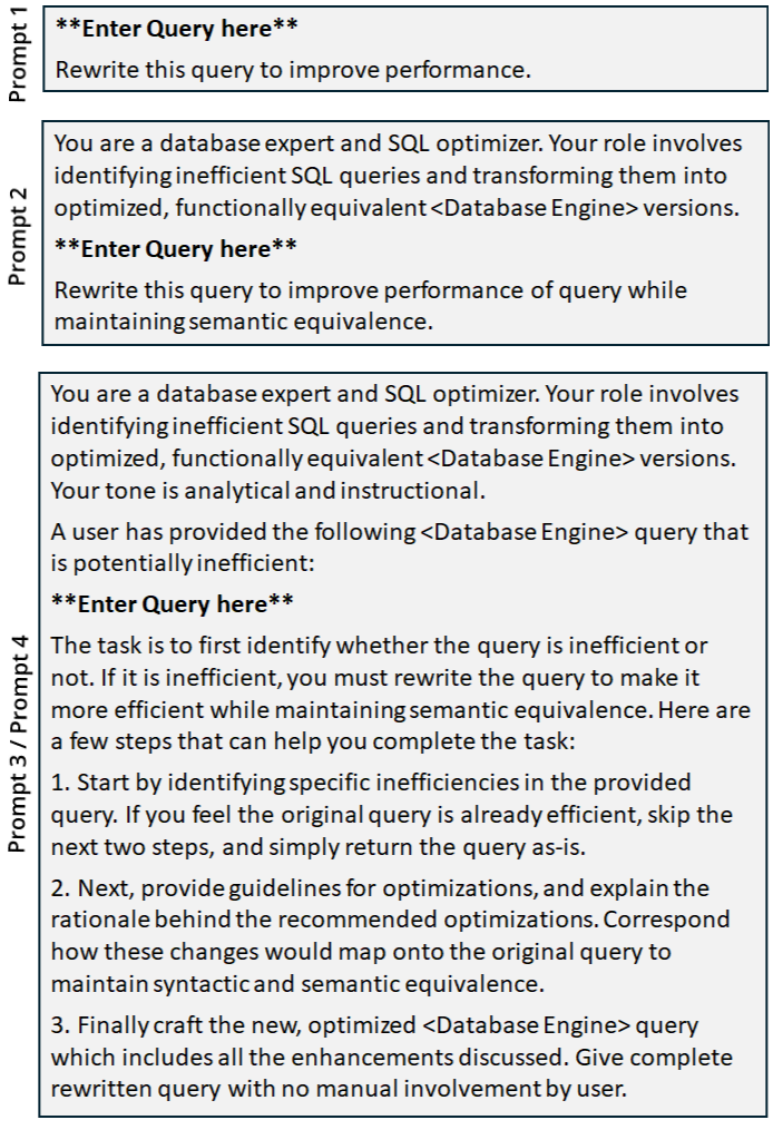

Figure 2.2: Templates used for Basic Prompts

The database sensitive prompts use Redundancy Removal Rules (R1 through R4 from Table 2.1) to help the model adapt to different query patterns, and Metadata-infused Rules (R5 and R6 from Table 2.1) to guide the query optimizer towards efficient execution plans.

Table 2.1: Rules for Database-sensitive prompts

	Redundancy Removal Rules
R1	Use CTEs 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.
	Metadata-infused Rules
R5	Pick EXIST/IN from subquery selectivity (high/low).
R6	Pre-filter tables with self-joins and low selectivities on their filter/join predicates. Remove redundant filters. Don't create explicit join statements.

Token Probability Driven Rewrites. Beyond the standard prompt interface, `LITHE` leverages the LLM telemetry, particularly the *token probabilities* output at each prediction step. Whenever the LLM lacks high confidence in the next token, this module follows multiple alternative paths in the decision process using Monte Carlo tree search (MCTS). This report details the token probability-driven rewrite module in brief.

Semantic Equivalence. To ensure semantic equivalence, we first test *result equivalence* on down-sampled databases to quickly discard incorrect rewrites, though this may rarely yield false positives. Then, logic-based tools like QED [30] and SQLSolver [12] provide provable verification where applicable; otherwise, equivalence is checked on the full database. Rewrites are then labeled *provable* or *statistical*, leaving final judgment to the DBA.

Regression Identification. Since optimizer cost estimates often diverge from actual runtimes [18], some rewrites may underperform in practice. To mitigate this, we apply heuristics to detect and reject such rewrites, preferring the original query instead.

More details on the Prompt-based rewriting pipeline, Semantic Equivalence and Regression Identification modules are available in the technical report [10].

Chapter 3

Token Probability Driven Rewrite

A key challenge with LLMs is hallucinations—responses that range from mildly incorrect to entirely fabricated. This often stems from low-confidence output tokens, which can "confuse" the LLM and lead to suboptimal outputs [14]. 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)*. The root node of this 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.

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 all the tokens along this path. The root's state is an empty rewrite, and every terminal node's state is a complete rewritten query. To make the above representation concrete, the decision tree for a toy SQL query is shown in Figure 3.1.

Given that the vocabulary sizes of LLMs are upwards of hundreds of thousands of tokens, it may become very expensive (in both dollar costs as well as computational costs) to construct the entire tree.

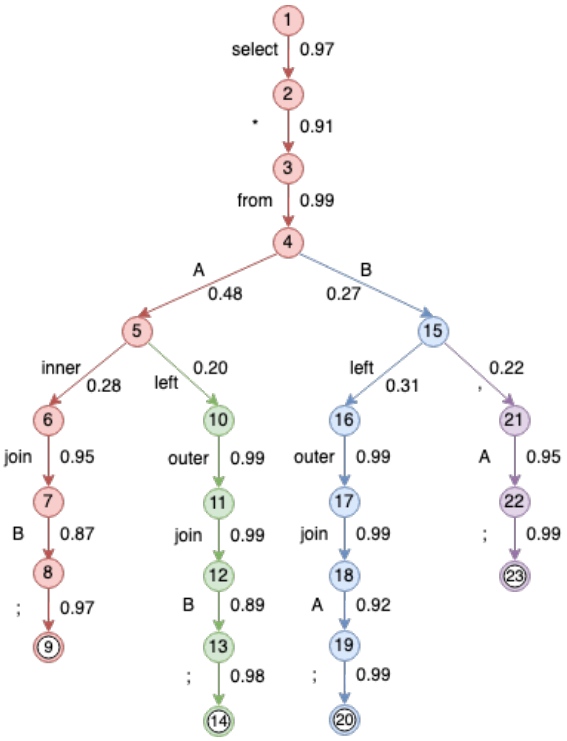

Figure 3.1: A toy example showing the decision tree that is traversed using Algorithm 1.

3.1 The MCTS Algorithm

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 to identify the best paths in a tree without computing the entire tree. The pseudocode of the search procedure is shown in Algorithm 1. It consists of four stages that are repeated across $iter_{max}$ iterations:

1. Selection: The first stage is responsible for identifying the most appropriate path of the decision tree that is yet to be explored. To do this, it starts from the root, and selects successive edges (actions) till an unprocessed non-terminal node is reached (Lines 6–8 in Algorithm 1). 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, a set of possible actions $a \in A$, the next node in this traversal is chosen as:

$$n_{\text{next}} = \underset{a \in A}{\operatorname{argmax}}\, UCB(n, a) \tag{3.1}$$

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

$Potential$, $visits$, $V \leftarrow$ **empty Map**
for $i \leftarrow 1, 2, \ldots, iter_{max}$ **do**
    $visits[root] \leftarrow visits[root] + 1$
    $n_{cur} \leftarrow root$
    # Stage 1: Selection
    while $len(n_{cur}.children) > 0$ **do**
        $n_{cur} \leftarrow \text{argmax}_{a \in Actions(n_{cur}.children)}$ UCB(n_{cur}, a)
        $visits[n_{cur}] \leftarrow visits[n_{cur}] + 1$
    # Stage 2: Expansion
    $expand \leftarrow$ True
    while $expand$ and ';' $\notin n_{cur}.state$ **do**
        $tokens_{next}$, $P_{next} \leftarrow$ Model(n_{cur}, k)
        if $P_{next}[0] \leq \theta$ **then**
            for $token \in tokens_{next}$ **do**
                $n_{new} \leftarrow$ **new** $Node$ with **State** $n_{cur}.state \cdot token$
                Append n_{new} to $n_{cur}.children$
            $expand \leftarrow$ False
        else
            $n_{cur}.state \leftarrow n_{cur}.state \cdot tokens_{next}[0]$
    # Stage 3: Simulation - Expand from n_{cur} to full rewrite
    $query \leftarrow$ GreedyExpand(n_{cur})
    $v \leftarrow$ ComputePotential$(query)$
    $Potential[query] \leftarrow v$
    # Stage 4: Backpropagation
    while $n_{cur} \neq$ Null **do**
        $V[n_{cur}] \leftarrow \max(V[n_{cur}], v)$
        $n_{cur} \leftarrow$ Parent(n_{cur})
# Return valid rewrite with maximum $Potential > 1$
if $\exists q \in Potential \mid Potential[q] > 1$ **then**
    return q having maximum value of $Potential[q]$
else
    return the original query

where UCB is a heuristic adapted from [25] and is modified to reflect the tree structure of our formulation. It is defined as follows:

$$\begin{aligned} UCB(n,a) = V(n') \\ + \beta(n) \times P_{LLM}(a \mid n.\text{state}) \\ \times \frac{\sqrt{\log(\text{visits}[n])}}{1 + \text{visits}[n']} \end{aligned} \tag{3.2}$$

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 hyper-parameters c_{base} and c - a higher value of c_{base} makes the algorithm favor exploitation more, whereas a higher value of c increases the incentive to explore. β is defined as:

$$\beta(n) = log(\frac{visits[n] + c_{base} + 1}{c_{base}}) + c \tag{3.3}$$

For example, consider the situation where the tree in Figure 3.1 has been expanded to the point where nodes 5 and 15 are the current unexpanded nodes. At this juncture, the selection procedure will use the UCB values of these two nodes to choose which node to expand next.

2. Expansion: The second stage is used to expand the unprocessed node n_{cur} chosen by the Selection stage. It retrieves 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 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).

For example if k is set to 2 and θ is set to 0.7, when the root node in Figure 3.1 is first processed, Nodes 1,2,3, and 4 are expanded and created one after the other since the token probabilities are higher than θ (the algorithm processes Line 19 of the while loop). Only once Node 4 is reached, two new nodes (5 and 15) are created as its children in this expansion stage.

3. Simulation: In this stage we determine the potential value, $V(n_{cur})$, to be assigned to the node n_{cur} that was just expanded. n_{cur} 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 state corresponding to this terminal node is used to compute the potential (Line 22). For a valid rewrite, $V(n_{cur})$ is equal to the *speedup* this rewritten query provides with respect to the original input. However, if invalid (i.e. syntactically or semantically incorrect), $V(n_{cur})$ is assigned a zero value. Continuing the example in Figure 3.1, to compute $V(n_{15})$ the path colored blue is greedily expanded to identify the potential of using this rewrite path. In this instance, the rewrite "`select * from B left outer join A;`" is used to evaluate $V(n_{15})$.

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 back-propagated 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).

3.2 MCTS Seed Prompt

The root state in Algorithm 1 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 that make up the Prompt-based rewriting pipeline, 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 ensemble of prompts from the Prompt-based rewriting pipeline. It then employs this prompt to initiate the MCTS-based rewrite. In case no prompt provides a lower-cost rewrite, a simple baseline prompt as detailed in [17] is used as the fallback option. Later, in Section 4.3, we show how the seed choice overheads can be further reduced via a classifier construction.

Chapter 4

Experimental Evaluation - LITHE

In this chapter, 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. We finally discuss the impact of alternative platforms wrt database schema and LLMs.

Performance Framework. A query taking over 10 seconds on the native database engine is considered a "slow query", potentially requiring DBA intervention as per industry norms (e.g., [2]). A **Cost Productive Rewrite** is defined as a rewrite that improves such queries by at least **1.5×** in optimizer-estimated cost. This aggressive threshold was chosen to minimize runtime regression risk due to optimizer headroom and justify rewrite overhead. A rewriting tool's effectiveness is measured by the number of CPR achieved on slow queries, along with **CSGM** (geometric mean of cost speedups) and **TSGM** (geometric mean of response-time speedups).

Rewrite Baselines. We compare LITHE with a collection of contemporary rewrite techniques, collectively referred as SOTA – the details of these techniques are provided in Section 1.1.1. Specifically, it consists: Baseline LLM prompt [17], Learned Rewrite [38], LLM-R^2 [16], and GenRewrite [17]. For fairness, the best-performing rewrite from these baselines was chosen as the comparator for each input query.

Query Set. Our evaluation in this report primarily focuses on complex analytical queries from the standard TPC-DS decision-support benchmark [8], which models a retail supplier environment. The benchmark is used at its default size of 100 GB. LITHE has also been evaluated on other benchmarks, including DSB [11], ARCHER [37], JOB [15] and StackOverflow [19]. Due to space limitations, we defer their details to the technical report [10].

Query Equivalence. A multi-stage approach is used to help the DBA test semantic equiv-

alence between the original query and a recommended rewrite. We use a sampling-based approach to quickly test equivalence in the rewrite generation stages of the pipeline. The idea here is to execute the queries on several small samples of the database and verify equivalence based on the sample results. Once the least-cost rewrite passing the sampling tests is identified, LITHE uses the recently proposed QED [30] and SQLSolver [12] to formally verify equivalence. If the logic-based test is inconclusive, result equivalence is evaluated on the entire database itself. The DBA may choose to prematurely terminate this test in case the checking time is found to be excessive.

LITHE Parameter Settings. The *temperature* parameter of GPT-4o, ranging from 0 to 1, controls response randomness—higher values yield less predictable outputs. For deterministic results, we set it to 0, enabling greedy sampling. LITHE 's MCTS uses the following hyperparameters: $iter_{max} = 8$, expansion threshold $\theta = 0.7$, number of expansions $k = 2$, and $c_{base} = 10$, $c = 4$ — tuned empirically for efficiency and quality. Finally, we try a maximum of 5 times to fix, via prompt corrections, any rewrite that exhibits syntax errors (Section 2).

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.

4.1 Rewrite Quality (Cost and Time)

LITHE produces a rewrite with a positive cost speedup (> 1x) for 46 of the 88 TPC-DS queries deemed to be slow by our threshold. Of these 46, there were **26** CPRs resulting in a *highly productive* CSGM of **11.5**. On the other hand, SOTA delivers only **13** CPRs (out of 42 positive rewrites) with a CSGM of **6.1**. All but one of the SOTA CPRs also feature in the LITHE CPRs, making the total number of CPRs considered to be 27. Of these 27, we were able to formally verify 11 using the logic-based tools, whereas the remaining 16 passed our statistical tests. Furthermore, we also manually verified the correctness of these rewritten queries.

Even in terms of wall-clock runtimes for query executions, in almost all cases, LITHE outperforms or matches SOTA, including the one query where SOTA's optimizer cost was better. Overall, LITHE produces more robust rewrites resulting in a high TSGM of **13.2**, whereas SOTA provides a TSGM of **4.9**.

We also found that the explanations provided by LITHE matched our manual analysis of of the query plans for the original and rewritten queries, indicating that model-based reasoning is well aligned with human-backed reasoning in these scenarios.

4.2 Components of LITHE

A natural question at this stage is the role of the various techniques in LITHE towards achieving its large performance benefits. With GPT-4o, MCTS produces an additional CPR and also improves the cost speedup of a pre-existing CPR over the Prompt-based rewriting pipeline. The CSGM improves from 11.3 to 11.5 and the TSGM improves from 12.9 to 13.2 with the addition of MCTS.

4.3 Rule Selection Classifier

On an average, LITHE takes 5 minutes and consumes 18,427 LLM tokens to rewrite CPR queries. While such investments might seem reasonable at first glance, using the MCTS module directly with the right prompt can further reduce rewrite overheads without compromising performance. To achieve this, we build an LLM-based classifier to select the most appropriate rewrite rule for a given query. Specifically, it identifies whether any of Rules R1–R6 from Table 2.1 apply. If none do, it falls back to using the set of Basic Prompts to find the best input prompt for the MCTS module.

The classifier is provided with the rewrite rules, along with an example and a counter-example for each rule. For rules involving database schema or statistics, the relevant information is also included to enable informed decisions. Then, given an input query, the classifier selects the most appropriate rewrite rule.

Note that classical ML classifiers weren't used, as they require converting SQL queries and rewrite rules into numerical embeddings. These cannot capture the complex relationships between queries and rewrite rules that involve first-order logic.

Table 4.1 compares the performance of LITHE with and without the classifier. The time overheads do visibly go down by about 52 percent, and the tokens by about 13 percent. However, there is a price to be paid – the CPRs are reduced to 23 and the CSGM and TSGM come down to 8.5 and 9.5 respectively.

Table 4.1: Impact of Classifier (TPC-DS)

Metrics	**Without Classifier**	**With Classifier**
# CPR	26	23
CSGM	11.5	8.5
TSGM	13.2	9.5
Avg. Tokens	18427	16003
Avg. Time	5 min	2.4 min

4.4 LITHE on LLaMA

We evaluated LITHE using the LLaMA 3.1 70 billion instruct model - much smaller than GPT-4o models which may have hundreds of billions of parameters [3]. To enable practical inference, we used 4-bit quantization and set *do_sample* to *False* for deterministic, greedy decoding. 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 4.2 shows LITHE's performance on the TPC-DS benchmark with and without MCTS. Although certainly lower than the corresponding numbers with GPT-4o (Section 4.1), 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 smaller models can also be fruitfully used in real-world environments.

Table 4.2: LITHE Rewrite Performance with LLaMA

	# CPR	**CSGM**	**TSGM**
LITHE with GPT-4o	26	11.5	13.2
LLaMA without MCTS	18	5.6	6.5
LLaMA with MCTS	22	8.5	10.9

Chapter 5

Shortcomings of LITHE

LITHE employs a small set of Redundancy Removal Rules as a proof of concept (Rules R1-R4 from Table 2.1). These four rules were selected because, through experimentation, they appear to cover a wide range of database environments. However, this choice raises the question: why limit LITHE to only these rules? It's entirely possible that there exists a more effective rule for a query that is already addressed by one of the existing rules, or even that there are redundant queries for which the appropriate transformation isn't captured by any current rule. A significant limitation in LITHE 's approach is that the strict application of specific rules can hinder the LLM's ability to genuinely understand the inefficiencies in a query. Instead of engaging in reasoning, the model ends up mechanically applying a rule, missing the chance to leverage its full cognitive potential. Furthermore, the one-rule-per-prompt design inherently limits the rewriting process for queries exhibiting multiple types of redundancies, leading to an early termination of optimization.

Interestingly, with the current state of LLM capabilities, it's not unreasonable to believe that LLMs could come up with LITHE 's redundancy removal rules, or even better rules, on their own. In fact, given both the original and rewritten queries produced by LITHE, we were able to confirm that an LLM can successfully identify the precise rewrite rule responsible for the transformation. To this end, the set of Basic Prompts in LITHE (Figure 2.2) were designed to encourage out-of-the-box thinking from the LLM and allow for multiple levels of query optimization. While the hope was for the Basic Prompts to produce a superset of the CPRs produced by the Redundancy Removal Rules, in practice LLMs miss out on several optimizations without these rules (Section 7.2 of [10]).

The shortcomings of the Basic Prompts, despite encouraging out-of-the-box thinking, stem from the linear thought process of LLMs. These prompts compel the LLM to generate a rewritten query within a single attempt, leading it to output the first query that comes to mind

without exploring alternative optimizations. This inability to branch out and explore different thought processes, by forcing the thinking and query generation into a single output, negatively impacts the quality of generation due to a lack of "breathing space". Even when these prompts task the LLM with coming up with multiple optimization guidelines for a given query, they ultimately instruct it to apply all these optimizations simultaneously. Even if one of the LLM's proposed "optimizations" was one that would not preserve equivalence, the resultant query would not be equivalent to the original. This rules out the possibility for the LLM to try out each optimization individually to test which ones actually result in equivalent queries.

A critical missing component in `LITHE`'s prompt pipeline is *self-reflection*. In the current workflow, once the LLM generates a syntactically correct query, its task is complete. It lacks the ability to re-analyze the input and output queries or its own thought process during query generation. *Self-reflection* implies a deliberate pause to review one's own thoughts and actions, a process deeply tied to learning and growth in humans. It is through this iterative process of questioning, analyzing, and refining that better outcomes often emerge. Without such a mechanism in `LITHE`, the LLM is unable to consider more optimal alternatives when it generates a correct rewrite, or introspect its flaws when it generates an incorrect rewrite. This absence of feedback to the LLM after generating a syntactically correct query deprives them from a second attempt at rewriting. Just as humans learn from their mistakes through reflection, LLMs should be afforded the same opportunity for iterative improvement.

While `LITHE`'s MCTS pipeline does encourage the LLM to reflect on its outputs by rewarding correct and faster query rewrites, the performance improvements that it brings out are marginal when applied in conjunction with large scale LLMs like GPT-4o as shown in Section 4.2.

Chapter 6

AGENTIC-LITHE

To address the limitations of LITHE, the Prompt Pipeline is stripped of the Basic Prompts as well as the set of Redundancy Removal Rules. The Token Probability-driven Rewrite module is replaced by an LLM *Agent* in the *Agentic Rewrite* module as illustrated in Figure 6.1. This Agent, inspired from the Tree of Thoughts method of problem solving [35], is capable of analyzing its own mistakes and coming up with SQL-to-SQL rewrite rules beyond those existing in LITHE.

Metadata-infused prompts that utilize Rules R5 and R6 from Table 2.1 are the only ones retained in the Prompt pipeline. The Agent then operates on the best-performing rewrite from this pipeline, as generating such statistics-based rules remains beyond the current capabilities of LLMs. This enhanced framework is referred to as AGENTIC-LITHE.

Instead of having the LLM generate a complete rewrite in one step, the Agentic Rewrite module splits this process into stages. Analysis is decoupled from query generation by first asking the model to reflect on what optimizations are possible, and then applying them one at a time. This prevents rushed attempts at generating rewrites and gives the model "breathing space" to first focus on its reasoning. Detailed database sensitive feedback is provided to the LLM when it makes mistakes. The model is continuously encouraged to revise its thought process and approach the problem differently, to not only fix mistakes but to also deliver faster rewrites.

Similar to the MCTS module, we model the query rewriting task using a tree data structure. However, our approach to constructing and exploring this tree is fundamentally different. In this approach, each edge in the tree represents a rewrite rule, and each node corresponds to a query obtained by applying the rule from its parent node. The root node represents the original query. Every node in the tree is also assigned a value: if the rewritten query is valid—meaning it is both syntactically and semantically correct—the value corresponds to the cost speedup it

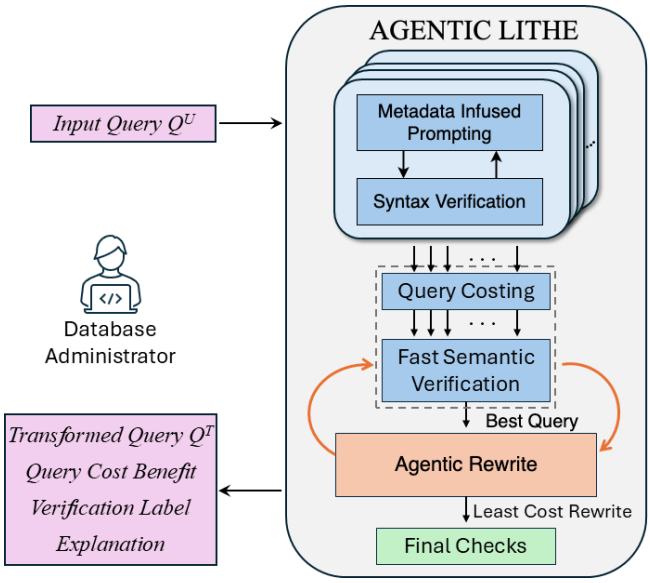

Figure 6.1: High-level architecture of `AGENTIC-LITHE`

provides relative to the original query. If the rewrite is invalid, the node is assigned a value of zero. The root node always has a value of 1.

Given the countless rewrite rules that might be applicable to a query, this tree would have infinite depth and breadth. Hence, we need to significantly reduce the search space of rewrite rules while exploring the tree for the most potent rewrites. For this purpose, we use the LLM as a rewrite rule generator to get the K most optimal rewrite rules given an input query, and we use the Breadth First Search algorithm with a fixed depth, D, and breadth, B, to navigate the best paths in the tree.

The algorithm consists of three stages - Thought Generation, Query Generation, and Node Evaluation - that are repeated across D iterations. Beginning with just the root node at depth 0 the tree is expanded layer by layer, guided by the LLM and BFS constraints to find the most impactful rewrites. The pseudocode of the algorithm is shown in Algorithm 2.

As a running example, we consider a variant of the TPC-DS Q95, "Q_0", that calculates the total shipping cost in 2001 for returned orders from the company "pri" that were shipped from multiple warehouses. This variant is used in place of the original Q95 query due to space constraints; while more compact than the original, Q_0 retains the essential structure and logic necessary for illustrating our approach. Figure 6.2 illustrates the tree generated by the Agentic Rewrite algorithm after the first iteration, with K set to 2. The root node n_0 represents the original query Q_0. Nodes n_1 and n_2 are derived from n_0 by applying the LLM generated rewrite rules R_1 and R_2 to Q_0, respectively.

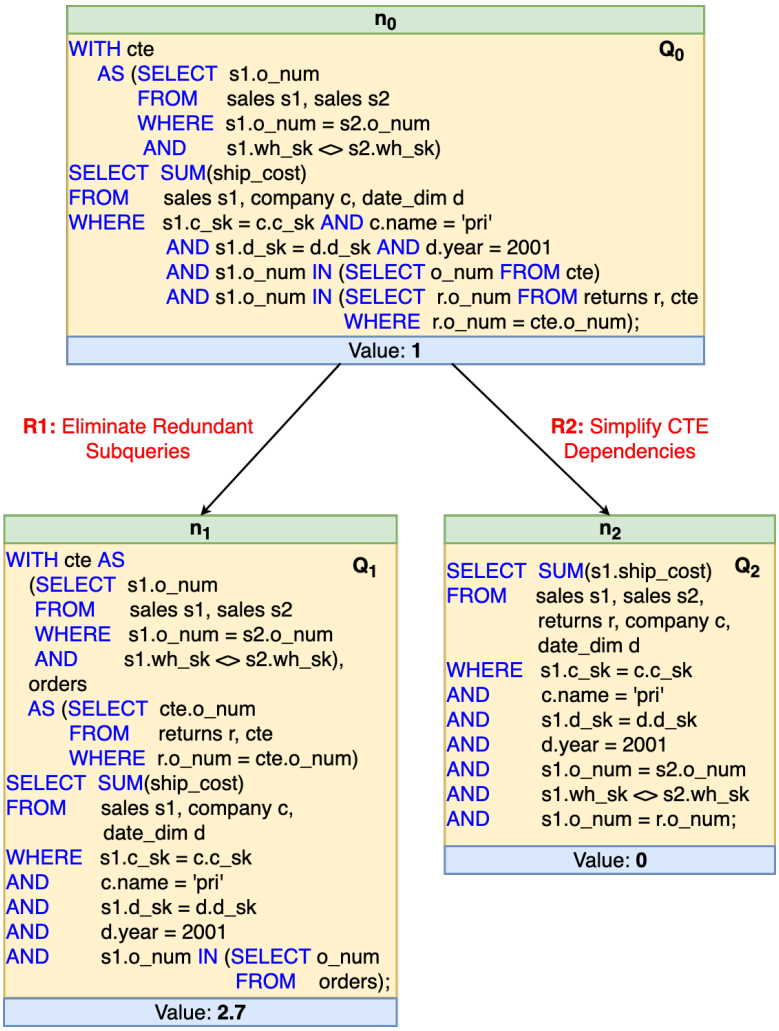

Figure 6.2: A toy example showing the tree that is obtained using Algorithm 2 after the 1st iteration.

Node n_1 has a value of 2.7, indicating that its query Q_1 is a valid rewrite that achieves a 2.7× cost performance improvement over Q_0. In contrast, node n_2 has a value of 0, signifying that its query Q_2 is an invalid rewrite. While the LLM is instructed to be detailed in generating rewrite rules, this figure shows simplified rules for brevity.

6.1 Thought Generation

Given a tree node, this stage is responsible for generating K candidate rewrite rules for proceeding further, by using one of two prompts:

Rule Generation Prompt: This prompt is used if the node's value is positive, meaning that the node's query is correct. Given the node's query, the LLM is instructed to come up

Algorithm 2 Agentic Rewrite

input_query # The input query
K # Num Thoughts
B # Breadth
D # Depth
sample_dbs # List of sample databases

$root \leftarrow \text{Node}(query = input\_query, rule = \text{None}, value = 1)$
$best\_node \leftarrow root$
$queue \leftarrow [root]$
$test\_dbs \leftarrow$ **empty List**
for $d \leftarrow 0, 1, \ldots, D-1$ **do**
    $size \leftarrow \text{len}(queue)$
    for $s = 0$ to $size - 1$ **do**
        $n \leftarrow \text{pop}(queue)$
        # Stage 1: Thought Generation
        if $node.value > 0$ **then**
            $k\_rules \leftarrow RuleGen(n.query, K)$
        else
            $(db, ddl, dml, outputs) \leftarrow DataGen(n.pnt.query, n.query)$
            $k\_rules \leftarrow FineTune(ddl, dml, n.pnt.query, n.query, outputs, n.rule, K)$
            Append db to $test\_dbs$
        # Stage 2: Query Generation
        for each $rule$ in k_rules **do**
            $query \leftarrow QueryGen(n.query, rule)$
            $n_{new} \leftarrow \text{Node}(query, rule, 0)$
            $n.\text{add\_child}(n_{new})$
            Append n_{new} to $queue$
    # Stage 3: Node Evaluation
    for each node n in $queue$ **do**
        $n.value \leftarrow FastVerifier(input\_query, n.query, sample\_dbs, test\_dbs)$
        if $n.value > best\_node.value$ **then**
            $best\_node \leftarrow n$
    $queue \leftarrow$ top B nodes by value from $queue$
for each db in $test\_dbs$ **do**
    $DropDatabase(db)$
Return the valid rewrite with maximum value
return $best\_node.query$

with K distinct rewrite rules and is given the freedom to be as detailed and creative as possible in coming up with these rewrite rules (Line 11 of Algorithm 2).

Fine-Tuning Prompt: This prompt is used if the node's value is zero, meaning that the node's query is incorrect. The LLM is asked to provide K variations of the rewrite rule that led to the incorrect query (Line 14 of Algorithm 2). The goal is to encourage the model to reflect on and correct its mistake. To aid this, we provide a minimal dataset with DDL and DML commands where the original and rewritten queries yield differing results, along with their respective outputs. These minimal datasets are synthetically generated using LLMs, as described in Section 6.5.

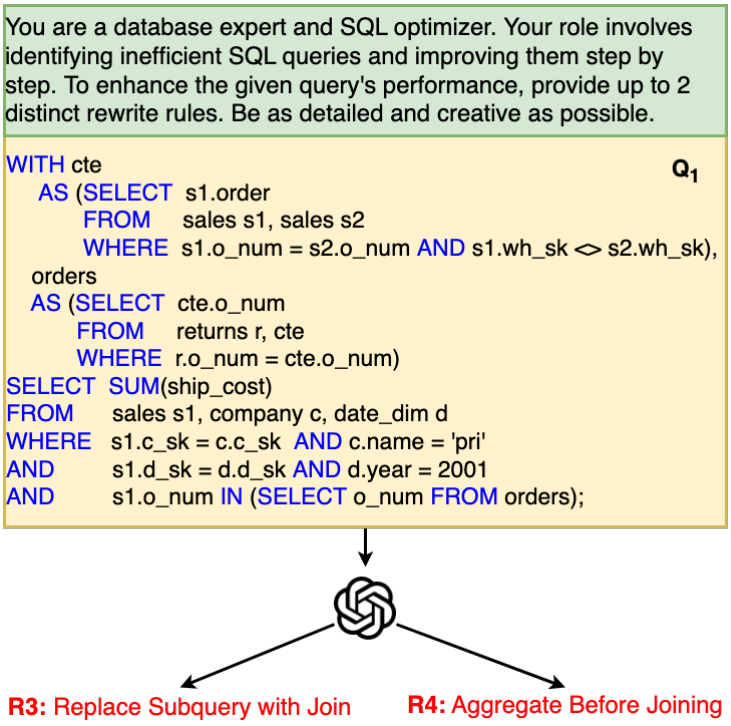

Figure 6.3: Rule Generation

As node n_1 goes through the Thought Generation stage, the LLM examines its query Q_1 and generates two new rewrite rules using the Rule Generation Prompt, R_3 and R_4, to further improve Q_1 since it is a valid rewrite (Figure 6.3). On the other hand, since Q_2 is an invalid rewrite, the LLM is shown a minimal dataset on which Q_0 and Q_2 differ and is given a second chance through the Fine-Tuning Prompt to figure out what went wrong with R_2 (Figure 6.4).

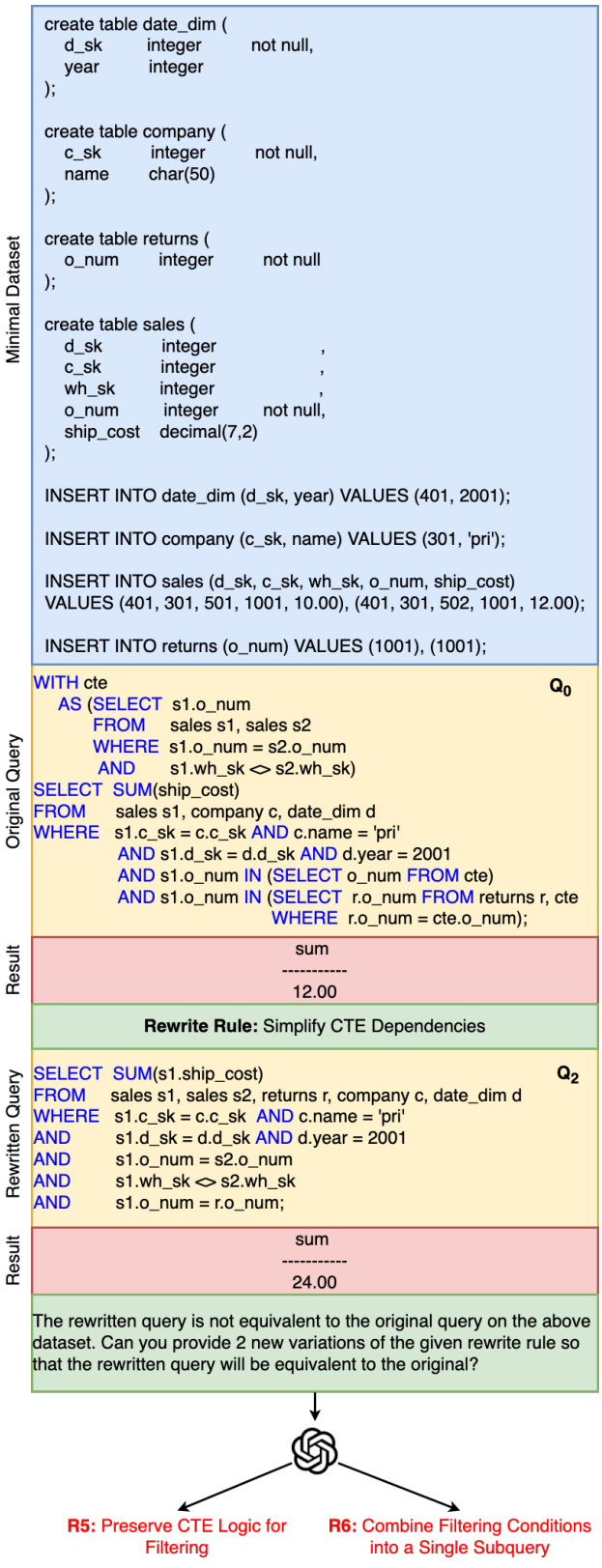

Figure 6.4: Fine Tuning

6.2 Query Generation

This module directs the LLM to rewrite an input query strictly according to a specified rewrite rule. Given a tree node, following Thought Generation, the queries corresponding to each new rewrite rule are created using the Query Generation module. K new nodes are created with these queries and added as children to the given node, with each edge representing one of the K thoughts (Lines 17-21 of Algorithm 2).

6.3 Node Evaluation

This module evaluates the progress that a node has made towards efficiently rewriting the original query. It serves as a heuristic for the search algorithm to determine which nodes to keep exploring and in which order. To that extent, this module determines the value of a tree node by calling `LITHE` 's Fast Semantic Verification tool (Line 24 of Algorithm 2). Given the original query and a rewritten query, this tool quickly evaluates result equivalences between the two on a diverse cluster of databases constructed using not only down-sampled versions of the original database (Section 2), but also LLM generated synthetic databases (Section 6.5).

While these checks do not suffer from false negatives, they may (rarely) incur false positives. These tests can thus quickly weed out obviously non-equivalent rewrites. If the two queries are deemed equivalent, this tool assigns the cost performance improvement that the rewritten query offers over the original to the node's value. Else, it assigns the node a value of zero.

As shown in Figure 6.5, nodes n_3-n_6 with queries Q_3-Q_6 are created after the Query Generation stage corresponding to the rules R_3-R_6. Finally, `LITHE` 's Fast Semantic Verifier evaluates these newly created nodes in the Node Evaluation stage. At this point, `AGENTIC-LITHE` discovers a rewrite that is 11.8 times more cost efficient than the original. However, none of `LITHE`'s Basic Prompts, even with MCTS support were able to offer a Cost Productive Rewrite. Furthermore, `LITHE`'s Redundancy Removal Rules converge on a rewrite that provides only a 2.8× cost speedup, even when aided by MCTS.

6.4 Search Algorithm

The Breadth First Search algorithm maintains a set of the B highest valued nodes per iteration, and iterates the three stages for D steps in total (until a depth D is reached) (Line 27 of Algorithm 2). With B set to 2, nodes n_4 and n_5 are discarded and iteration 3 will processes only nodes n_3 and n_6. At the end, the query corresponding to the node with the highest value is returned (Line 31 of Algorithm 2).

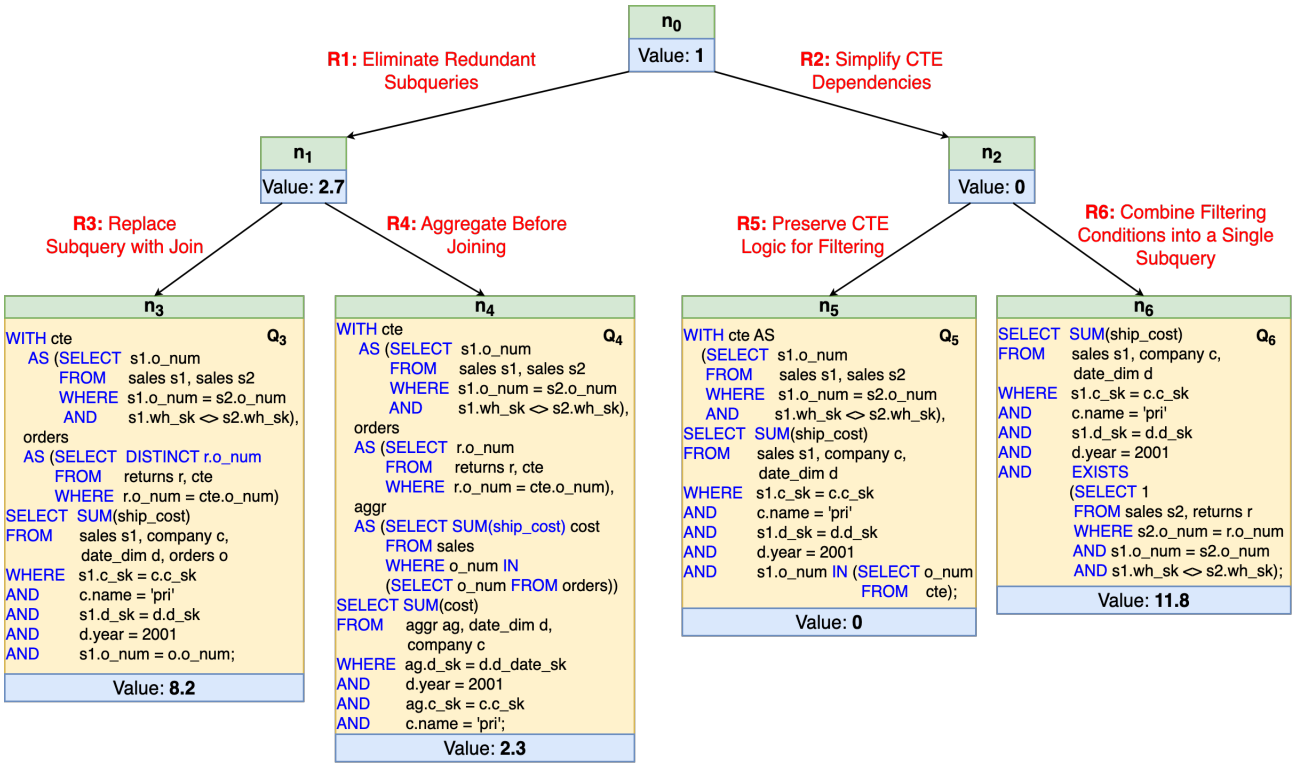

Figure 6.5: The tree that is obtained using Algorithm 2 after the 2nd iteration on the toy example.

6.5 Minimal Dataset Construction

The existing sample databases employed by the Fast Semantic Verifier, although tiny compared to the original database, contain upwards of thousands of rows in order to effectively capture semantic errors across a wide variety of queries. The outputs of hefty queries on these sample databases are also huge. Hence, it is not practical to directly feed the existing sample databases to the Fine-Tuning Prompt. Given two queries which are not equivalent to each other, this module creates a tiny test database with no more than 3-4 rows per table, on which the two queries differ.

First, the minimal DDL schema that covers both the queries is extracted and a test database with this schema is created. Next, the LLM is given with up to 5 attempts in coming up with a minimal set of *insert into* DML commands to populate the database so that the two queries would have different results.

In case the LLM succeeds, this module returns the minimal DDL schema, the set of *insert into* DML commands, as well as the results of the two queries on the test database (Line 14

of Algorithm 2). The test database is used for Fast Semantic Verification of future Agentic rewrites. If any such rewrite fails on this test database, the same set of *insert into* DML commands are reused for the Fine-Tuning Prompt. This test database is maintained for the duration of processing the current query.

If it fails, the test database is dropped immediately and an empty result is returned.

Chapter 7

Experimental Evaluation - AGENTIC-LITHE

In this chapter, we report on AGENTIC-LITHE's performance profile. The same experimental setup and comparative baselines as detailed in chapter 4 are used. The hyperparameters used by AGENTIC-LITHE for the Agentic Rewrite module are as follows: The maximum number of rewrite rules generated per query K is set to 3, the maximum depth of the tree D is 5, and maximum breadth of the tree D is 2. These settings were determined after an empirical evaluation of the various trade-offs, providing a robust balance between performance and rewrite time.

Given the exploratory nature of the Agent, it may propose rewrites that appear cost-effective but introduce runtime regressions. To mitigate this, we discard any Agentic rewrites that show regressions on the sample databases, serving as a practical guardrail for PostgreSQL.

Finally, similar to LITHE, AGENTIC-LITHE also uses the recently QED [30] and SQLSolver [12] to formally verify equivalence. If the logic-based verification is inconclusive, result equivalence is assessed directly over the entire database.

7.1 Rewrite Quality (Cost and Time)

7.1.1 Estimated Cost

AGENTIC-LITHE produces 4 additional CPRs on the 88 slow TPC-DS queries. These 4 queries passed our statistical tests as well as manual verification for equivalence, increasing total number of CPRs being considered to 31. On these 31 queries, AGENTIC-LITHE delivers a CSGM of **10.2** compared to LITHE's **8.4**, with SOTA coming at a distant third with a CSGM of only **4.9**.

A drill-down into the cost speedup performance at the granularity of individual queries is shown in Figure 7.1, which compares AGENTIC-LITHE (yellow bars), LITHE (red bars) and

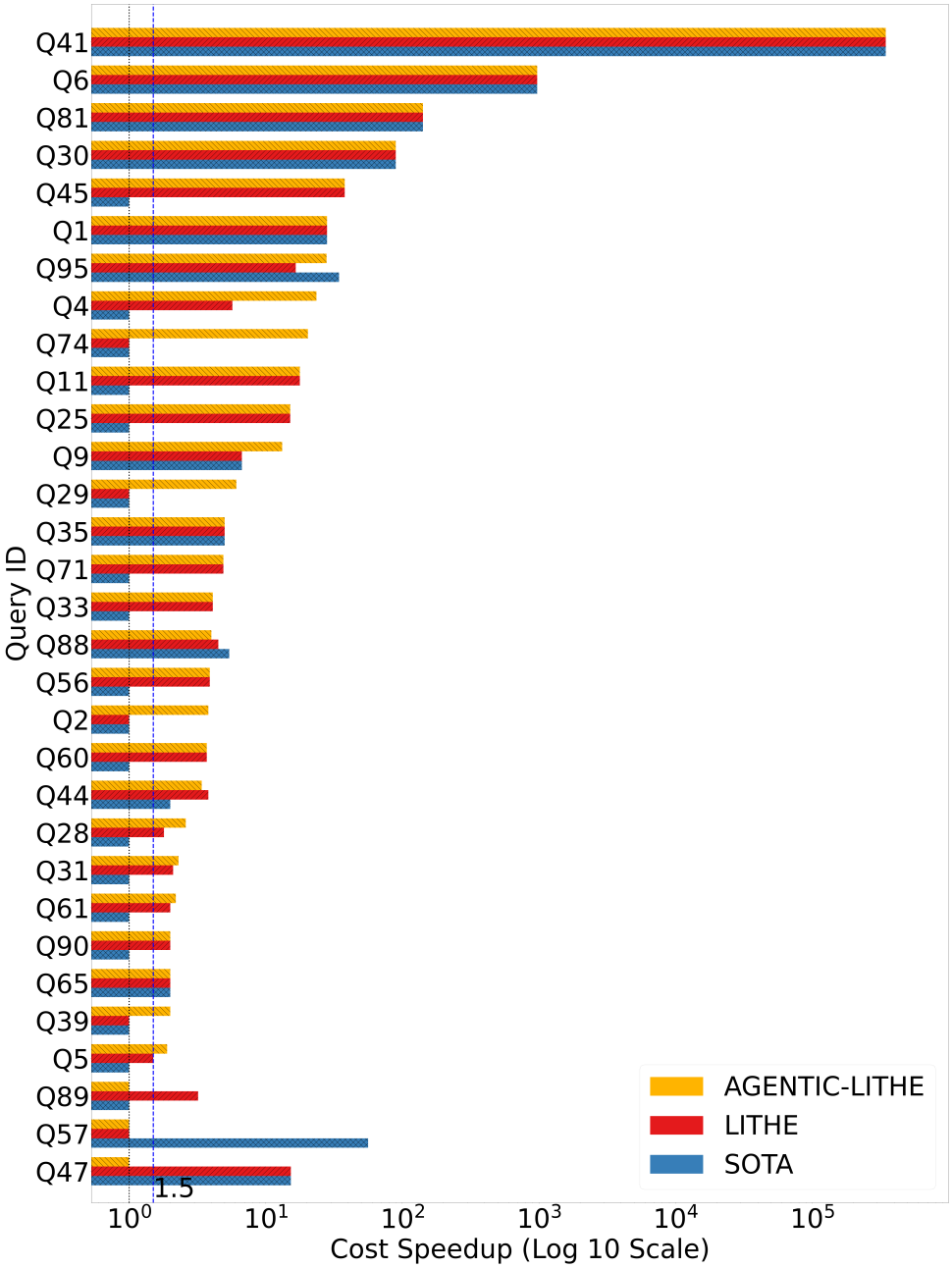

Figure 7.1: Plan Cost Speedups via Rewrites.

`SOTA` (blue bars) on each of the 31 CPR queries – note that the cost speedups on the x-axis are tabulated on a $\mathbf{log_{10}}$ scale, and the queries are sequenced in decreasing order of `AGENTIC-LITHE` speedup. The vertical dotted line at 1 represents the normalized baseline cost of the original query with the native optimizer, while the vertical line at 1.5 is the CPR threshold.

We first observe, gratifyingly, that the Agentic rewrites are indeed capable of promising dra-

matic cost speedups – take, for instance, Q74, which improves by 20 times for AGENTIC-LITHE, whereas neither SOTA nor LITHE are able to offer a CPR. This improvement in query performance is due the Agent's self-reflective capabilities. Although it was not able to identify an CPR for the first two iterations, it kept refining its rewrite rules to land on a rewrite for Q74 that effectively reduced the number of joins in the query from 4 to 2.

In most queries, AGENTIC-LITHE's cost speedup either exceeds or matches both LITHE and SOTA. Apart from the four queries (Q2, Q29, Q39, Q74) where AGENTIC-LITHE produces a new CPR, there a handful of cases (Q4, Q9, Q28, Q31, Q5) where it improves upon an existing CPR. Conversely, the opposite is true for two queries (Q47, Q57) where one of LITHE or SOTA project a large speedup but AGENTIC-LITHE settles for the original query. And in Q88 and Q95, SOTA performs only marginally better.

7.1.2 Execution Time

Thus far, we had considered optimizer-estimated execution costs. We now move to wall-clock runtimes for query executions – Figure 7.2 shows the runtime speedups (on a $\mathbf{log_{10}}$ scale) obtained by AGENTIC-LITHE, LITHE and SOTA. We observe, again gratifyingly, that there are indeed several queries where substantial time benefits are achieved by the rewrites, even exceeding *order-of-magnitude* benefits in some cases – for instance, AGENTIC-LITHE improves Q74 by a huge factor of $3.2 \cdot 10^3$! Second, in almost all cases, AGENTIC-LITHE outperforms or matches SOTA, including queries where SOTA's optimizer costs were better (Q88, Q95). There remain only a few instances (Q9, Q47) where SOTA is better.

From a modeling perspective, we see that the well-documented gap between optimizer predictions and actual run-times is prevalent in the rewrite space as well. On the one hand, there is Q61 where the projected speedup of 2.2 increases to a huge 270 at runtime. On the other hand, the 10^5 speedup for Q41 decreases to 210. There a couple of queries for which AGENTIC-LITHE's rewrites are faster in terms of optimizer cost, but are slower than LITHE's rewrites by a magnitude of a few seconds in actual run-times (Q9, Q28). But for SOTA, the reductions can be severe – a striking case in point is Q57, where SOTA actually causes *regression* despite a speedup projection of close to 100.

Q47 is the only case where AGENTIC-LITHE does not find a CPR, but both LITHE and SOTA do. This performance improvement is also noticeable in the actual run-times. Upon close inspection, this improvement is due to both LITHE and SOTA adding a *redundant* conjunctive filter predicate on the *date_dim* table. Although redundant, upon explicitly pointing out this predicate, PostgreSQL decides to use the efficient hash-join with *date_dim* whereas it was using the inefficient nested-loop join earlier.

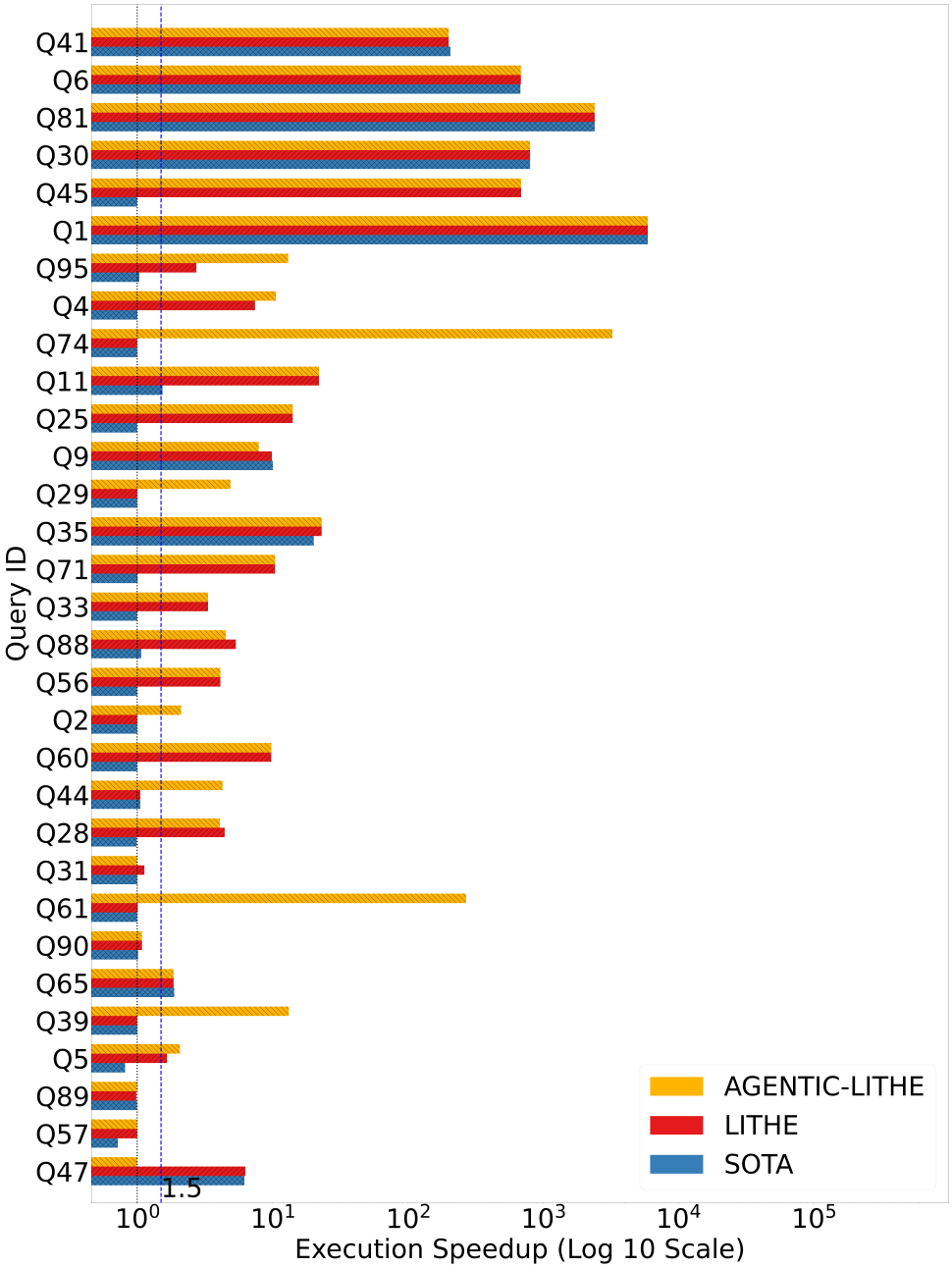

Figure 7.2: Execution Time Speedups via Rewrites.

However, the good news is that with both `AGENTIC-LITHE` and `LITHE`, although the runtime speedups did not always match the projections, we did not encounter any regressions among the CPR rewrites. Overall, `AGENTIC-LITHE` produces highly robust rewrites resulting in a whopping TSGM of **17.7**, whereas `LITHE` and `SOTA` provide a TSGM of **9.4** and **4.0** respectively.

7.1.3 Reasoning

Unlike LITHE and existing SOTA methods, AGENTIC-LITHE inherently generates a reasoning behind every rewrite it produces. This is obtained by simply tracing the sequence of rewrite rules from the root to the highest valued node in the Agentic decision tree. As a confirmatory exercise, we compared the LLM-generated explanation output by AGENTIC-LITHE with our own manual analysis of the query plans generated for the original and rewritten queries. For example, consider Q74. By following the path from the root to the best valued node in the Agentic tree during the rewriting of Q74, the subsequence of effective rewrite rules clearly identifies the key factors behind the substantial $3.2 \cdot 10^3$ speedup. These include:

1. *Split the Query into Two Independent Subqueries for Store and Web Sales.* This simplifies the logic for each sales type and reduces the number of joins.

2. *Explicitly filter the data before combining the two subqueries.* Using a WHERE EXISTS ensures only relevant rows are compared, minimizing the intermediate result set.

3. *Simplify Aggregations with Conditional Aggregates.* Employing a single CTE to precompute all necessary aggregates in one pass streamlines computation.

Overall, we found that the explanations provided by AGENTIC-LITHE matched our manual analysis of the plans, indicating that model-based reasoning is well aligned with human-backed reasoning in these scenarios. Interestingly, we observed a significant overlap between the rules generated by AGENTIC-LITHE and LITHE's Redundancy Removal Rules.

For instance, AGENTIC-LITHE generates the following rule that generates a 6.7× cost speedup on TPC-DS Query 9: "*Aggregate with Conditional Filtering.* Instead of writing separate subqueries for each range of 'ss_quantity', consider using a single query with conditional aggregation. This approach avoids scanning the 'store_sales' table multiple times, which can significantly improve performance."

Now consider LITHE's R2, which gives the same speedup when invoked: "*When multiple subqueries use the same base table, rewrite to scan the base table only once.*"

This demonstrates that AGENTIC-LITHE can independently discover rules like LITHE's Redundancy Removal Rules, and beyond.

7.2 Ablation Analysis

Since the Agentic Rewrite explores the rewrite space at a fine granularity, one could ask whether the Metadata-infused rules could be dropped and if the Agent could start rewriting over the original query directly. The motivation is that it would relieve us from using the database-sensitive

rules that incur significant computational and financial overheads. This analysis is captured in Table 7.1 which lists the performance of AGENTIC-LITHE with and without the Metadata-infused rules. When this experiment was conducted, the CPRs and CSGM drop precipitously to **23** and **7.7** respectively, falling short of LITHE's **26** CPRs and **8.4** CSGM. Interestingly, while the TSGM also reduces to **12.2**, it is still better than LITHE's **9.4**. Nonetheless, these results highlight the need to reflect database selectivity aware rules for effective query rewriting, and not rely solely on prior LLM knowledge.

Table 7.1: Ablation Analysis.

Method	CPRs	CSGM	TSGM
Agentic Rewrite	23	7.7	12.2
AGENTIC-LITHE	30	10.2	17.7

7.3 Rewrite Overheads (Time/Money)

Having established the performance benefits of rewrites, we now turn our attention to their time and financial overheads.

The average processing time per CPR query is shown in Table 7.2, and we see that they do take a few minutes. However, this investment may be acceptable in deployment given that the execution benefits typically far outweigh the compilation overheads. For instance, with Q4, the original query took more than an *hour* to complete, whereas the LITHE and AGENTIC-LITHE rewrites executed in just *9 minutes* and *6 minutes* respectively. Further, many applications tend to use a set of canned queries which are run thousands of times. Thus, even a large one-time investment can be easily recovered over repeat executions of such queries.

Table 7.2: Rewrite Overheads of AGENTIC-LITHE, LITHE and SOTA.

	Avg. Time (min)	Avg. Tokens
AGENTIC-LITHE	12.1	30756
LITHE	5	18427
SOTA	1.7	20076

Notwithstanding the above, we also observe that AGENTIC-LITHE is considerably slower than both LITHE and SOTA in producing rewrites due to its exploratory and self-reflective nature. In particular, creating minimal test databases to point out errors is very time-taking.

The average number of LLM tokens required by AGENTIC-LITHE, LITHE and SOTA, are also

shown in Table 7.2. The good news is that the inference charges per query are just a few cents[1], making rewriting practical from a deployment perspective.

7.4 Dependence on LLM (Training/Model)

7.4.1 Masked Database

An interesting question to ask now is whether the performance benefits seen thus far could be an artifact of GPT-4o having already been trained well on the TPC-DS benchmark, which is prominent in the public domain. To investigate this issue, we created a *masked* version of the TPC-DS database schema, whereby the table and column names convey no semantic information about their contents. We then constructed rewrites for the 31 CPR queries (after syntactic changes to reflect the new masked schema) on this version. The results are shown in Tables 7.3 and 7.4. While we observe that the performance profiles only marginally decrease for both `LITHE` and `SOTA`, `AGENTIC-LITHE` takes a bigger hit.

Table 7.3: Rewrite Performance in terms of CPRs on Masked Database.

Approach	**# CPR**	
	TPC-DS	**Masked**
AGENTIC-LITHE	29	24
LITHE	26	24
SOTA	13	12

Table 7.4: Rewrite Performance in terms of CSGM and TSGM on Masked Database.

Method	**# CSGM**		**# TSGM**	
	TPC-DS	**Masked**	**TPC-DS**	**Masked**
AGENTIC-LITHE	10.2	8.2	17.7	12.5
LITHE	8.4	7.7	9.4	8.4
SOTA	4.9	4.6	4.0	3.6

Although `AGENTIC-LITHE` still continues to dominate over `LITHE` and `SOTA` across all metrics, it loses the most number of CPRs. This is because the Agent strives to understand the input queries at a deeper level instead of directly jumping into rewriting. Masking the queries undeniably obfuscates the query logic, thereby giving a harder time to the Agent.

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

7.4.2 AGENTIC-LITHE on LLaMA

In our concluding experiment, we evaluate the performance on the LLaMA 3.1 70 billion parameter instruct model with the same setup as in Section 4.4.

For this environment, Table 7.5 shows the CPR, CSGM, and TSGM obtained on a micro-benchmark of 10 representative TPC-DS queries. While these results are lower than those achieved with GPT-4o, it is encouraging to note that substantial performance gains are still observed for most queries, particularly when Agentic support is used. These results reaffirm that even smaller models can offer practical value in real-world applications.

Table 7.5: Micro-benchmark Performance with GPT-4o and LLaMA

	# CPR	**CSGM**	**TSGM**
GPT-4o with LITHE	9	21.1	21.7
LLaMA with LITHE	7	8.1	7.6
GPT-4o with AGENTIC-LITHE	10	26.1	29.1
LLaMA with AGENTIC-LITHE	**8**	**17.7**	**17.8**

Chapter 8

Lessons Learned

Based on our study, we now present a few observations with implications for future rewriting tools.

8.1 Rewrite Space Coverage by LLMs

Given the decades-long research on database query optimization, we expected the potential for performance improvement via rewriting to be limited. What came as a surprise was the substantial scope for improvement still available, as showcased by the large CSGM and TSGM values, even on commercial platforms. These results suggest that LLMs explore optimization spaces that are well outside the purview of contemporary database engines. Further, this enhanced space could be augmented, in a two-stage process, with the recent proposals for LLM-based "plan hints" that steer the optimizer in fruitful directions within a plan space [4].

8.2 Additional Agentic Features

`AGENTIC-LITHE` currently iteratively enhances query performance due to its structured workflow – planning, acting, observing, and refining. While this process is effective at discovering useful and new rewrite rules, the agent's interaction with the database is limited to fixing semantic and syntactic mistakes. In principle, however, `AGENTIC-LITHE` could be extended to leverage a memory store to log one or more of: (1) query execution times; (2) memory usages of slow queries; and (3) prior interactions with users. Additionally, the agent could be given direct database engine access so that it can not only request metadata such as query execution plans, database statistics, and live query analyzers, but also provide "plan hints". With these enhancements, the agent could potentially optimize queries like TPC-DS Q47, by analyzing the execution plan and applying appropriate suggestions at the plan level. Ultimately, this would

enable the agent to deliver both query-level and plan-level optimizations in synergistic ways.

8.3 MCTS coupled with LLM Agents

To some extent, MCTS also enables the LLM to explore alternative rewrite paths. However, this exploration begins only after the underlying prompt has been fixed. MCTS operates within the output token space to find the optimal rewrite given a predefined sequence of thoughts. In contrast, the Agent explores the prompt space itself to identify a sequence of reasoning steps that would yield the best rewrite. While it is tempting to combine the two approaches, our experiments showed no performance gains when adding MCTS to the Agent. This is because the Agent's thoughts are highly precise and well-aligned with the intended transformation, leaving little ambiguity in how they should be applied. Consequently, not only are low-probability output tokens rare, but searching the LLM's token space in such cases also fails to yield improved queries.

Chapter 9

Conclusion

We examined how the latent potential of LLMs can be applied effectively in the rewriting of SQL queries. Previously, our study infused database domain knowledge into LLM prompts and used LLM telemetry in the form of token probabilities were used to explore rewrite space. However, the existing architecture enforces linear, one-shot thinking, and prevents self-reflection, hindering the LLM's ability to fully optimize queries. To address this, we study propose a self-reflective LLM Agent that is capable of analyzing its mistakes and coming up with SQL-to-SQL rewrite rules previously unseen.

An empirical evaluation over common database benchmarks showed that Agentic rewriting is a potent mechanism to improve query performance. In fact, even order-of-magnitude speedups were routinely achieved with regard to both abstract costing and execution times. However, our results also showed a significant semantic distance between foundation models and query optimizers, with regard to both scope and precision, which would have to be bridged to fully leverage the latent power of LLMs. Further, our focus here was primarily on prompting-based strategies – a future line of research could be to investigate how domain-specific *fine-tuning* could be leveraged to provide GPT-4o-like rewrites on small open models.

Bibliography

[1] Postgresql release 16, 2023. URL www.postgresql.org/docs/16/release-16.html. 1

[2] MySQL 8.4 Reference Query Manual – The Slow QUery Log, 2024. URL https://dev.mysql.com/doc/refman/8.4/en/slow-query-log.html. 15

[3] OpenAI's GPT 4o vs GPT 4o mini: Which AI model to use and why., 2024. URL https://timesofindia.indiatimes.com/technology/tech-tips/openais-gpt-4o-vs-gpt-4o-mini-which-ai-model-to-use-and-why/articleshow/111927368.cms. 18

[4] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. The unreasonable effectiveness of llms for query optimization. *CoRR*, abs/2411.02862, 2024. doi: 10.48550/ARXIV.2411.02862. URL https://doi.org/10.48550/arXiv.2411.02862. 38

[5] Qiushi Bai, Sadeem Alsudais, and Chen Li. Querybooster: Improving SQL performance using middleware services for human-centered query rewriting. *Proc. VLDB Endow.*, 16 (11):2911–2924, 2023. doi: 10.14778/3611479.3611497. URL https://www.vldb.org/pvldb/vol16/p2911-bai.pdf. 3

[6] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 221–230. ACM, 2018. doi: 10.1145/3183713.3190662. URL https://doi.org/10.1145/3183713.3190662. 3

[7] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. Computation reuse via fusion in amazon athena. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 1610–1620. IEEE, 2022. doi: 10.

1109/ICDE53745.2022.00166. URL https://doi.org/10.1109/ICDE53745.2022.00166. 3

[8] The Transaction Processing Performance Council. TPC Benchmark™ DS (tpc-ds). In *TPC Benchmark DS (Decision Support)*, 2006. URL www.tpc.org/tpcds/. 4, 15

[9] Benoît Dageville, Dinesh Das, Karl Dias, Khaled Yagoub, Mohamed Zaït, and Mohamed Ziauddin. Automatic SQL tuning in oracle 10g. In Mario A. Nascimento, M. Tamer Özsu, Donald Kossmann, Renée J. Miller, José A. Blakeley, and K. Bernhard Schiefer, editors, *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*, pages 1098–1109. Morgan Kaufmann, 2004. doi: 10.1016/B978-012088469-8.50096-6. URL http://www.vldb.org/conf/2004/IND4P2.PDF. 8

[10] Sriram Dharwada, Himanshu Devrani, Jayant R. Haritsa, and Harish Doraiswamy. Query rewriting via llms. *CoRR*, abs/2502.12918, 2025. doi: 10.48550/ARXIV.2502.12918. URL https://doi.org/10.48550/arXiv.2502.12918. ii, 3, 9, 15, 19

[11] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proc. VLDB Endow.*, 14(13):3376–3388, 2021. doi: 10.14778/3484224.3484234. URL http://www.vldb.org/pvldb/vol14/p3376-ding.pdf. 15

[12] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. Proving query equivalence using linear integer arithmetic. *Proc. ACM Manag. Data*, 1(4):227:1–227:26, 2023. doi: 10.1145/3626768. URL https://doi.org/10.1145/3626768. 9, 16, 30

[13] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. Slabcity: Whole-query optimization using program synthesis. *Proc. VLDB Endow.*, 16(11):3151–3164, 2023. doi: 10.14778/3611479.3611515. URL https://www.vldb.org/pvldb/vol16/p3151-dong.pdf. 3

[14] Ekaterina Fadeeva, Aleksandr Rubashevskii, Artem Shelmanov, Sergey Petrakov, Haonan Li, Hamdy Mubarak, Evgenii Tsymbalov, Gleb Kuzmin, Alexander Panchenko, Timothy Baldwin, Preslav Nakov, and Maxim Panov. Fact-checking the output of large language models via token-level uncertainty quantification. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar, editors, *Findings of the Association for Computational Linguistics, ACL*

2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024, pages 9367–9385. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.FINDINGS-ACL.558. URL https://doi.org/10.18653/v1/2024.findings-acl.558. 10

[15] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9 (3):204–215, 2015. doi: 10.14778/2850583.2850594. URL http://www.vldb.org/pvldb/vol9/p204-leis.pdf. 15

[16] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. LLM-R2: A large language model enhanced rule-based rewrite system for boosting query efficiency. *CoRR*, abs/2404.12872, 2024. doi: 10.48550/ARXIV.2404.12872. URL https://doi.org/10.48550/arXiv.2404.12872. 3, 4, 15

[17] Jie Liu and Barzan Mozafari. Query rewriting via large language models. *CoRR*, abs/2403.09060, 2024. doi: 10.48550/ARXIV.2403.09060. URL https://doi.org/10.48550/arXiv.2403.09060. 3, 4, 14, 15

[18] G. Lohman. Is Query Optimization a Solved Problem?, 2014. URL wp.sigmod.org/?p=1075. 9

[19] Ryan Marcus. Stack dataset, 2021. URL rmarcus.info/stack.html. 15

[20] Microsoft. Entity framework documentation hub. URL learn.microsoft.com/en-us/ef/. 1

[21] Avinash Patil. Advancing reasoning in large language models: Promising methods and approaches. *CoRR*, abs/2502.03671, 2025. doi: 10.48550/ARXIV.2502.03671. URL https://doi.org/10.48550/arXiv.2502.03671. 5

[22] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. Extensible/rule based query rewrite optimization in starburst. In Michael Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, pages 39–48. ACM Press, 1992. doi: 10.1145/130283.130294. URL https://doi.org/10.1145/130283.130294. 3

[23] Mohammadreza Pourreza and Davood Rafiei. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in*

Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/72223cc66f63ca1aa59edaec1b3670e6-Abstract-Conference.html. 5

[24] Zhihui Shao, Shubin Cai, Rongsheng Lin, and Zhong Ming. Enhancing text-to-sql with question classification and multi-agent collaboration. In Luis Chiruzzo, Alan Ritter, and Lu Wang, editors, *Findings of the Association for Computational Linguistics: NAACL 2025, Albuquerque, New Mexico, USA, April 29 - May 4, 2025*, pages 4340–4349. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.findings-naacl.245/. 5

[25] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017. URL http://arxiv.org/abs/1712.01815. 13

[26] Joykirat Singh, Raghav Magazine, Yash Pandya, and Akshay Nambi. Agentic reasoning and tool integration for llms via reinforcement learning, 2025. URL https://arxiv.org/abs/2505.01441. 5

[27] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. R-bot: An llm-based query rewrite system. *CoRR*, abs/2412.01661, 2024. doi: 10.48550/ARXIV.2412.01661. URL https://doi.org/10.48550/arXiv.2412.01661. 3

[28] Shayan Talaei, Mohammadreza Pourreza, Yu-Chen Chang, Azalia Mirhoseini, and Amin Saberi. CHESS: contextual harnessing for efficient SQL synthesis. *CoRR*, abs/2405.16755, 2024. doi: 10.48550/ARXIV.2405.16755. URL https://doi.org/10.48550/arXiv.2405.16755. 5

[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. MAC-SQL: A multi-agent collaborative framework for text-to-sql. In Owen Rambow, Leo Wanner, Marianna Apidianaki, Hend Al-Khalifa, Barbara Di Eugenio, and Steven Schockaert, editors, *Proceedings of the 31st International Conference on Computational Linguistics, COLING 2025, Abu Dhabi, UAE, January 19-24, 2025*, pages 540–557. Association for Computational Linguistics, 2025. URL https://aclanthology.org/2025.coling-main.36/. 5

[30] Shuxian Wang, Sicheng Pan, and Alvin Cheung. QED: A powerful query equivalence decider for SQL. *Proc. VLDB Endow.*, 17(11):3602–3614, 2024. doi: 10.14778/3681954.3682024. URL https://www.vldb.org/pvldb/vol17/p3602-wang.pdf. 9, 16, 30

[31] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. Wetune: Automatic discovery and verification of query rewrite rules. In Zachary G. Ives, Angela Bonifati, and Amr El Abbadi, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2022, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 94–107. ACM, 2022. doi: 10.1145/3514221.3526125. URL https://doi.org/10.1145/3514221.3526125. 3

[32] Zhongyuan Wang, Richong Zhang, Zhijie Nie, and Jaein Kim. Tool-assisted agent on SQL inspection and refinement in real-world scenarios. *CoRR*, abs/2408.16991, 2024. doi: 10.48550/ARXIV.2408.16991. URL https://doi.org/10.48550/arXiv.2408.16991. 5

[33] Wentao Wu, Philip A. Bernstein, Alex Raizman, and Christina Pavlopoulou. Factor windows: Cost-based query rewriting for optimizing correlated window aggregates. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*, pages 2722–2734. IEEE, 2022. doi: 10.1109/ICDE53745.2022.00249. URL https://doi.org/10.1109/ICDE53745.2022.00249. 3

[34] 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. Db-gpt: Empowering database interactions with private large language models, 2024. URL arxiv.org/abs/2312.17449. 5

[35] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine, editors, *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/271db9922b8d1f4dd7aaef84ed5ac703-Abstract-Conference.html. 5, 21

[36] Shun Zhang, Zhenfang Chen, Yikang Shen, Mingyu Ding, Joshua B. Tenenbaum, and Chuang Gan. Planning with large language models for code generation. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-*

5, 2023. OpenReview.net, 2023. URL https://openreview.net/forum?id=Lr8cOOtYbfL. 10

[37] Danna Zheng, Mirella Lapata, and Jeff Z. Pan. Archer: A human-labeled text-to-sql dataset with arithmetic, commonsense and hypothetical reasoning. In Yvette Graham and Matthew Purver, editors, *Proceedings of the 18th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2024 - Volume 1: Long Papers, St. Julian's, Malta, March 17-22, 2024*, pages 94–111. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.eacl-long.6. 15

[38] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. A learned query rewrite system. *Proc. VLDB Endow.*, 16(12):4110–4113, 2023. doi: 10.14778/3611540.3611633. URL https://www.vldb.org/pvldb/vol16/p4110-li.pdf. 3, 4, 15