Declarative Query Transformation Using LLMs: Database Aspects

A PROJECT REPORT
SUBMITTED IN PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
Master of Technology
IN
Faculty of Engineering

BY

Himanshu Devrani

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

June, 2025

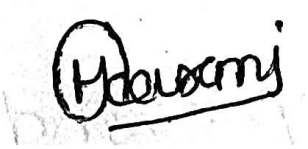
Declaration of Originality

I, **Himanshu Devrani**, with SR No. **04-04-00-10-51-23-1-23106** hereby declare that the material presented in the thesis titled

Declarative Query Transformation Using LLMs: Database Aspects

represents original work carried out by me in the **Department of Computer Science and Automation** at **Indian Institute of Science** during the years **2024-2025**.
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

DEDICATED TO

The Student Community

who can use and reuse this template to glory

Acknowledgements

I would like to express my deepest gratitude to my advisor, **Prof. Jayant Haritsa**, at the Database Systems Lab, IISc, for his invaluable guidance, constructive feedback, and constant encouragement throughout my time at IISc. His expertise and insights have been instrumental in shaping this work.

I am also sincerely thankful to **Dr. Harish Doraiswamy**, Principal Researcher at Microsoft Research, Bangalore, for offering us the opportunity to pursue this project as part of an internship at Microsoft. His guidance and technical suggestions played a crucial role in transforming this work into a meaningful and impactful outcome. I also thank Microsoft Research for providing the computational resources necessary to complete this project.

I would like to extend special thanks to my project partner, **Sriram Dharwada**, a fellow M.Tech CSA 2025 student, for the countless brainstorming sessions, technical discussions, and collaborative effort that went into building this project together.

I am also grateful to the faculty and staff of the Department of Computer Science and Automation, IISc, and to the members of the Database Systems Laboratory, for fostering an environment of learning, collaboration, and innovation.

My heartfelt thanks go to my colleagues and friends for their encouragement and support during the challenging phases of this work.

I am especially thankful to my family for their unwavering support, patience, and belief in me throughout this journey.

Finally, I acknowledge the resources and infrastructure provided by the Database Systems Lab, which were instrumental in completing this work.

Abstract

Optimizing complex SQL queries by rewriting them into simpler, more efficient forms has long been a cornerstone of database performance tuning. Transforming inefficient SQL queries into optimized yet semantically equivalent versions can greatly improve database performance as well as developer's comprehensibility. Although significant progress has been made in developing rewriting techniques and query optimizers, certain challenges still persist. Traditional approaches primarily operate within the execution plan space, applying heuristic or cost-based transformation rules. While these strategies have proven effective in certain scenarios, they are often rigid, limited in scope, and struggle to generalize across diverse query patterns and database systems

This study explores how the advanced reasoning capabilities of LLMs can be utilized for efficient and reliable query rewriting directly at query level without going in to the plan space. We introduce a structured methodology that combines a foundational suite of generic prompts, along with database-aware prompts tailored for eliminating redundancies and applying selectivity-based rules.

A unique challenge in this setting is that, unlike tasks such as Text-to-SQL—where some degree of ambiguity is inherent—SQL-to-SQL rewriting involves transforming a precisely defined and unambiguous query. Therefore, there is no margin for semantic errors; the rewritten query must be logically and functionally identical to the original. To address this, our system incorporates a set of statistical and logic-based validation mechanisms that rigorously verify the correctness of each transformation. Furthermore, to bridge the gap between optimizer-estimated costs and actual runtime performance, we use an analytical framework to detect and mitigate any instances of potential runtime performance regression during the rewriting process itself.

Testing on industry-standard and real-world benchmarks shows our system outperforms SOTA techniques by an order of magnitude. For instance, with TPC-DS on PostgreSQL, 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. Beyond performance improvements, our

tool also excels in query readability metric, significantly performing better than SOTA. This LLM-driven solution functions as a reliable intermediary tool between enterprise applications and databases, ensuring efficiency and performance. It can also help upcoming database systems lacking high-quality optimizers cheaply circumvent their initial limitations. Notably, our experiments also reveal that our LLM-driven solution delivers substantial benefits even for the industrial-grade optimizers, further highlighting its versatility and robustness.

This study is a joint project with another M.Tech CSA student, Sriram Dharwada. In this thesis, I will focus on detailing the components and work implemented by me, while the remaining aspects of the project are covered in the technical report [16].

Publications based on this Thesis

1. 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

```
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

SQL queries in enterprise applications are often burdened with inefficiencies and unnecessary complexity, particularly when generated by tools like ORM frameworks. A clear illustration of this issue is the blog-processing query [32] presented in Figure 1.1, which was created using the widely used Entity Framework [32]. 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 (assuming `NOT NULL` column constraints and key-joins), as demonstrated in Figure 1.2.

```
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

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 [2], 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)[39], with their advanced context understanding, can interpret the query semantically and transform it into more efficient SQL instructions. Therefore, a 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 also empower new or open-source optimizers to achieve performance on par with commercial database optimizers, where modifying the optimizer itself is a complex and time-consuming task.

Building on this context, we have designed an effective SQL-to-SQL query transformer which meets the following essential criteria: (1) The transformed query should be semantically equivalent to the original; (2) The rewrite should ideally improve performance, but at least not cause regression; and (3) The transformation overheads must be practical for deployment.

1.1 Related Work

Rule-based SQL rewriting. Most of the recent work on SQL query rewriting is rule-based [49, 43, 10, 45, 12, 33]. For instance, WeTune [43] uses 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 [20, 30].

Learned Rewrite [49] uses existing Calcite [11] rules and aims to learn the optimal subset of rules along with the order in which they must be applied. Since the rewrite search space grows exponentially with the number of rules, it uses MCTS scheme to efficiently navigate this space and find the rewritten query with maximum cost reduction.

LLM-R^2 [29] 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 [36] 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 [10] implements human-centered rewriting – it provides an interface to specify rules using an expressive rule language, which it generalizes to create rewrite rules to be applied on the query. There are also rule-based rewrite approaches designed for specific types of rewrites such as optimizing correlated window aggregations [45] and common expression elimination [12].

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 [30] is the first LLM-based approach to use the LLM for end-to-end query rewriting. Instead of using predefined rules from Calcite [11], 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, LLMs can struggle to apply rules correctly if not provided with accompanying examples. 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 Database Modules. LLM technologies have been advocated for a variety of database modules. For instance, they have been extensively used for Text-to-SQL transformations [27, 46, 5, 4, 37, 50]. The main focus of these techniques is to correctly ascertain the information necessary to formulate the SQL query [40, 34, 37]. On the other hand, the goal of S2S 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.

In recent times, LLMs have also been considered for plan-hinting [9], join-order optimization [38], index selection [50], data pipelines [23], data management [26], etc. An LLM-enabled multi-modal query optimizer [42], where data spans text, image, audio and video domains, has also been proposed recently. These approaches can be used in conjunction with the `LITHE` system since they address orthogonal segments of the query processing pipeline.

1.2 The LITHE Rewriter

A recent vision paper [50] advocates the use of LLMs for such query rewriting. The state-of-the-art (SOTA) techniques have also foregrounded the potential benefits of using LLMs for query rewriting. However, these benefits are often limited due to: (a) a restricted scope of rewrites, (b) vulnerability to semantic and syntactic errors, and (c) reliance on plan-space transformations (i.e., optimizing over execution plan nodes) rather than operating directly in the query space (i.e., transforming the query structure itself).

To address these limitations, we present LITHE (LLM-Infused Transformation of Hefty Queries), an LLM-based query rewriter. LITHE employs a range of prompting strategies augmented with domain-specific hints and underlying database metadata to guide the LLM in rewriting queries. LITHE applies transformation directly in the query space which provides greater scope for candidate rewrites since only intent, and not implementation, is expressed, and this multiplicity of rewrites in turn results in a broader coverage of the underlying plan space.

We conducted a calibrated study to understand how different types of prompting strategies affect the LLM's rewriting capabilities. In addition, we incorporated specific guardrails to ensure that the rewrites remain semantically equivalent and beneficial in practical, real-world settings.

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. Incorporates database-sensitive rules in LLM prompts, covering both schematic and statistical dimensions.
4. Evaluates rewrite quality over a broad range of database environments, demonstrating substantial benefits over both SOTA and the native optimizer.

5. Identifies learnings that could help guide research directions for industrial-strength query rewriting.

1.3 Results

Our first set of experiments to evaluate LITHE's performance is carried out on the industry standard TPC-DS benchmark [14], 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 [49], LLM-R^2 [29], GenRewrite [30], as well as a baseline LLM prompt [30]). 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. In our second stage of experiments, we evaluate generalizability of the above outcomes in a variety of new scenarios, including (a) Additional benchmarks, (b) commercial database engines and (c) unseen database schemas.

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.

1.4 Organization

The remainder of this thesis is organized as follows: Chapter 2 presents the architecture and overview of LITHE. Chapter 3 discusses the different prompting strategies used in LITHE. Chapter 4 highlights LITHE's implementation choices. Chapter 5 provides a detailed experimental evaluation. Finally, Chapter 6 summarizes our conclusions and outlines directions for future research.

Chapter 2

LITHE Architecture

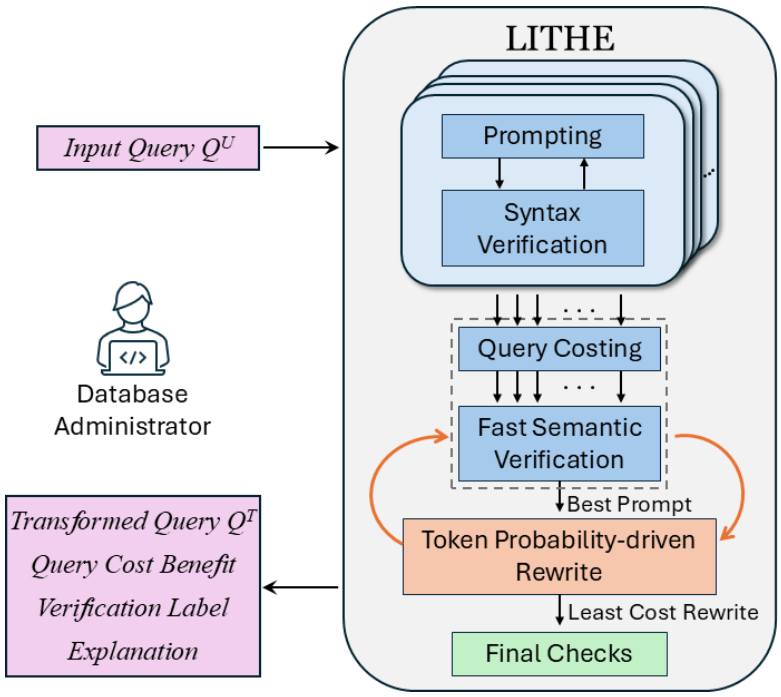


Figure 2.1: High-level architecture of LITHE

We propose 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. 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.

A query will be tagged as provable if it is verified by a logic-based tool that guarantees correctness; otherwise, if it passes verification on a sampled database, it will be marked as statistical.

Armed with this information, DBAs can leverage their expertise to decide whether or not to use Q^T. Note that having the DBA in the loop is a common practice in commercial query advisory systems [15].

2.1 LITHE Overview

The LITHE architecture, illustrated in Figure 2.1, consists of the following five-stage pipeline:

1. Prompt-based rewriting. This component consists of two modules: LLM Prompting and Syntax Verification. In the prompting module a user query with a crafted prompt is fed to LLM asking for rewrite. The different kind of prompts used in the LITHE are described below:

Basic Prompts. We begin with a 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 that the best prompt granularity is query-specific. Moreover, this basic prompt ensemble was found to itself 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. Our rules are invoked directly in *query space*, providing the LLM with 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 [11] rules) typically used in existing rewrite systems, which operate in *plan space*.

In particular, we work with two classes of rules – *1. Redundancy Removal Rules*: These rules eliminate repeated and redundant computations of the same output; and *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 in prompts 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.

The LLM is prompted using each of the different prompts or rules described in Chapter 3. Once a rewrite is generated by the LLM, it is passed to the database parser for syntax verification. If the rewrite is found to be syntactically correct, it is forwarded to the next module—the query costing module. Otherwise, the syntax error is fed back to the LLM for correction. To prevent runaway correction loops, we apply a threshold on the number of allowed syntax

correction attempts.

2. Query Costing. The costs of candidate rewrites are evaluated via the database engine's optimizer. Rewrites whose costs are greater than that of the original query are immediately discarded. Whereas, the potentially beneficial rewrites (if any) are checked for semantic equivalence to the original query.

3. Fast Semantic Equivalence. Statistical (result-based equivalence on sampled databases) techniques (as described in Section 4.2) are employed to quickly and cheaply assess the semantic equivalence of a recommended rewrite. If the rewrite is deemed valid by this module, it is returned along with the prompt that generated it; otherwise an invalid label is returned.

4. Token probability-driven rewrite. The prompt producing the most beneficial (and valid) rewrite is used as input to a Monte-Carlo tree search (MCTS)-based procedure to further refine the rewrite quality (details in techreport [16]). The query costing and semantic equivalence modules are also used internally within this procedure.

5. Final Checks and Output. Once a least expensive valid rewrite(as identified after the MCTS module) is obtained, it has to go through two checks:

1. Equivalence Check: The least expensive valid rewrite is evaluated using a suite of logic-based techniques (as described in Section 4.2) to assign a final equivalence label—either provable or statistical. If the rewrite is verified by the logic-based tool, it is labeled as provable; otherwise, it is considered statistical. Following this, the cost benefit of the rewrite over the original query is computed.

2. Regression Check: Further, the execution "brittleness" of least expensive rewrite is assessed using the robustness heuristics(as described in Section 4.1).

If no valid rewrite is identified, or if the rewrite is expected to be a regression, the original query itself is returned to the DBA. Whereas, if a beneficial rewrite is recommended, an LLM-generated reasoning for the expected performance improvement is also extracted.

2.2 Performance Framework

We consider a query that takes more than T seconds to complete on the native database engine as a "slow query", potentially triggering intervention by the DBA. Based on common industry perception (e.g. [3]), T is set at 10 seconds in our study. For this context, we define a **Cost Productive Rewrite (CPR)** as a rewrite that improves a slow query's performance by at least **1.5** times wrt the optimizer-estimated cost – this aggressive choice of threshold is so that: (a) there is enough headroom in the optimizer prediction that a runtime regression is unlikely; and (b) the benefits of the rewrite substantively outweigh the transformation overheads.

The overall benefit provided by a rewriting tool is quantified by the number of CPR obtained

on the slow query workload. Additionally, we also measure **CSGM**, the *geometric mean* (GM) of the cost speedups obtained by these rewrites. Finally, to assess the actual run-time benefits, we evaluate **TSGM**, the geometric mean (GM) of the response-time speedups obtained by these rewrites.

2.3 Query Micro-benchmark

To motivate the progression of prompting strategies incorporated in `LITHE`, we create an initial micro-benchmark comprising 10 diverse TPC-DS queries for which we were able to *hand-craft* high-quality CPRs. These queries are processed on GPT-4o, the popular OpenAI LLM, and the rewrites are evaluated on the PostgreSQL v16 database engine. The human rewrites deliver a CSGM of **11.84**, serving as an aspirational target to attain computationally. Later, in Chapter 5, we extend the evaluation to complete benchmarks.

Further, for ease of presentation, we focus on the CSGM metric in Chapter 3. The TSGM performance is subsequently discussed in Chapter 5.

Chapter 3

Prompting Strategies

In this chapter, we explore the simplest interface to LLMs, namely *prompting*, for query rewriting. Our work involved exploring a range of prompting strategies to understand their effectiveness in guiding LLMs for query rewriting. These strategies were grouped into two categories:

3.1 Basic Prompts

We evaluate four basic prompts, enumerated in Figure 3.1, which cover a progressive range of instructional detail and test the effectiveness of the LLM's base knowledge.

Prompt 1: This is the baseline prompt used in [30], 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 human experts.

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

Table 3.1 shows the performance of the four prompt templates on the micro-benchmark. We find that less than half the rewrites are productive with individual prompts. However, a drill-down shows that the best prompt in the ensemble is *query-specific* – this opens up the possibility

of using all four prompts in parallel, and then choosing the best among them. This ensemble approach raises the CPRs to **6** (Row 5 in Table 3.1) – however, there remain four queries that are not productively rewritten by these prompts.

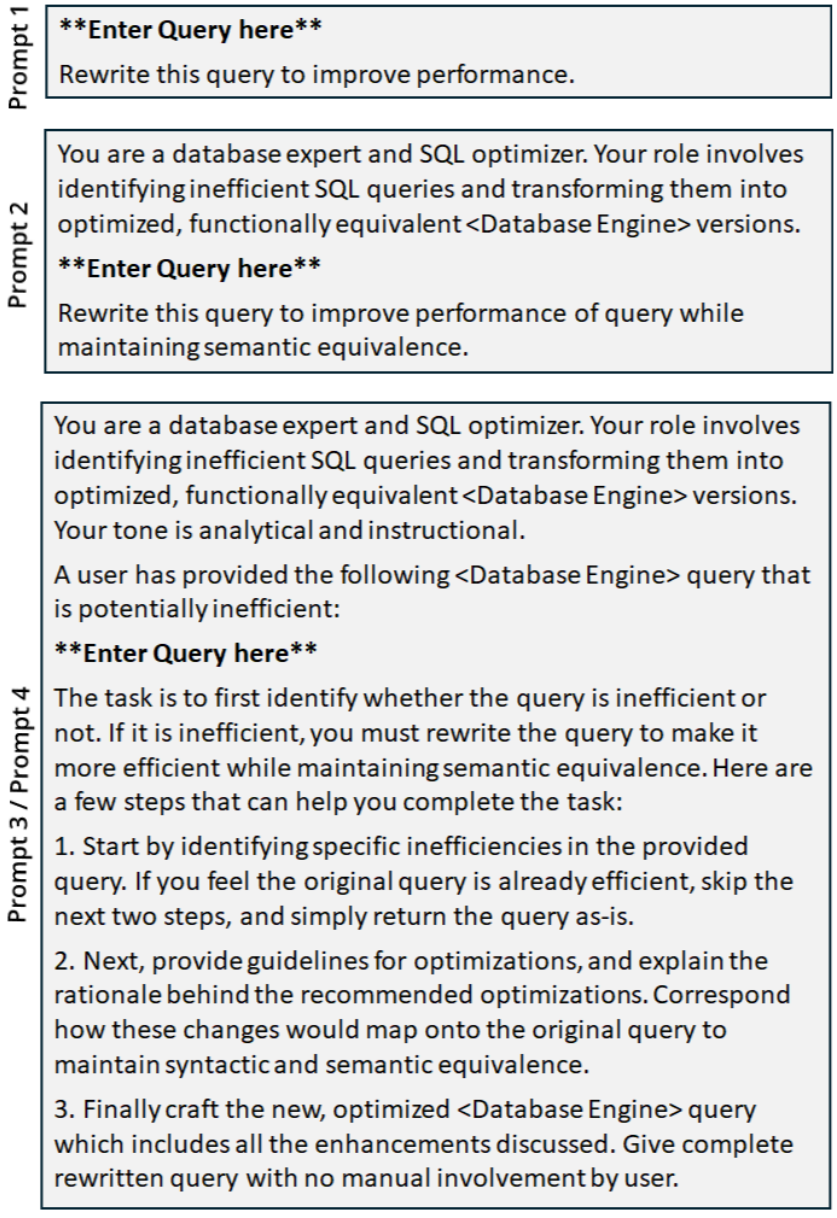


Figure 3.1: Templates used for Basic Prompts

The CSGM, shown in the last column of Table 3.1, is at most **3** for the individual prompts, while the ensemble reaches **3.23**. But these speedups, although productive, are all lower than those delivered by the human rewrites.

Prompt	# CPR	CSGM
Prompt 1	3	1.99
Prompt 2	2	1.85
Prompt 3	4	2.83
Prompt 4	4	3.00
Prompt Ensemble	6	3.23
SOTA Ensemble	3	2.49

Table 3.1: Performance of Basic Prompts on micro-benchmark.

Finally, an ensemble of `SOTA` techniques (described in Chapter 5) was also processed on the same platform. They delivered 3 CPRs with a CSGM of 2.49 (last row in Table 3.1), indicating the wide gap between the current reality and what is humanly possible.

3.2 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.

3.2.1 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 3.2 are designed to tackle such redundancies. The relevant schematic information (e.g. table names, column names, constraints) required by these rules is also provided in the prompt.

The template for such rule-based prompts is shown in Figure 3.2(a) and includes an example

	Redundancy Removal Rules
R1	Use CTEs (Common Table Expressions) to avoid repeated computation of a given expression.
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 tables involved in self-joins and with low selectivities on their filter and/or join predicates. Remove any redundant filters from the main query. Do not create explicit join statements.

Table 3.2: Rules for Database-sensitive prompts.

to demonstrate the rule application to the LLM – the specific examples used with our rules are available in Appendix. 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 3.1) is shown in Table 3.3. We observe that the CPR increases to **7**, and CSGM grows to **6.85**.

Prompt	**# CPR**	**CSGM**
Basic Prompts $\bigcup$ {R1, ..., R4}	7	6.85

Table 3.3: Performance with Redundancy Removal Rules on micro-benchmark.

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 introduces significant computational overheads due to the vast number of possible rule application orders, making it expensive to explore. So, for simplicity, we have chosen to process them individually rather than cumulatively.

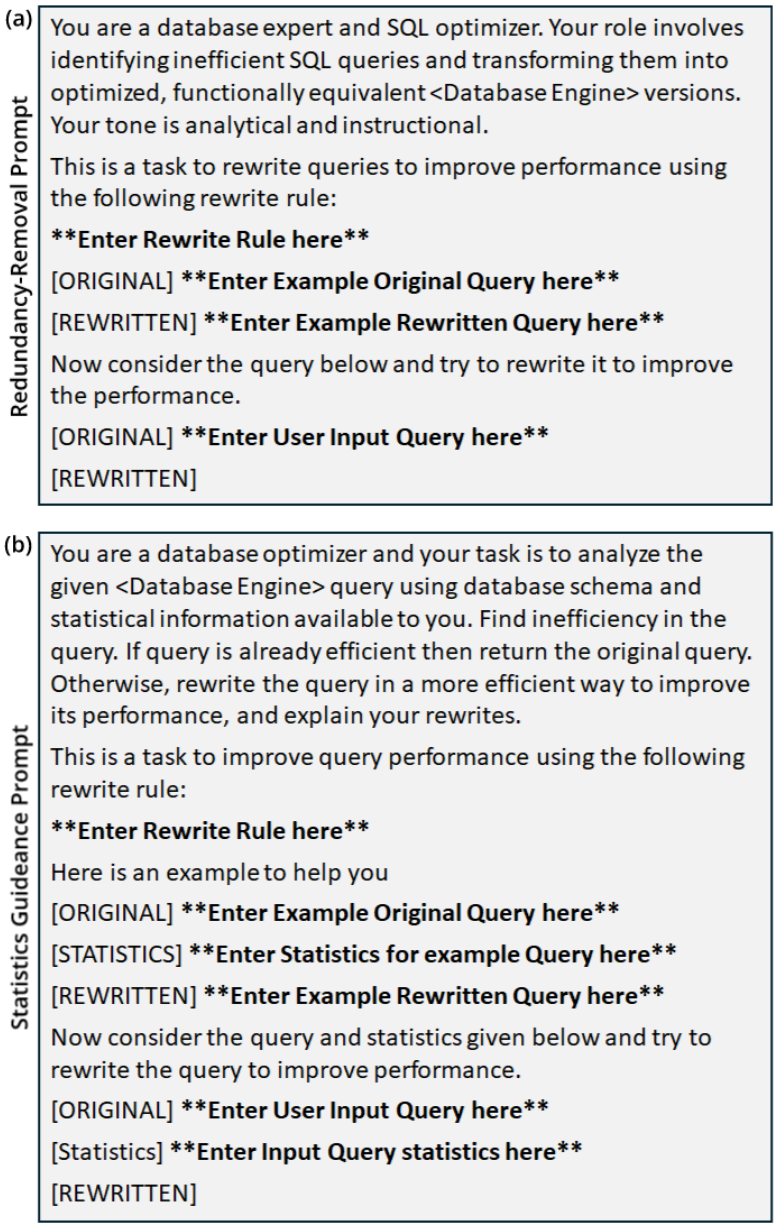


Figure 3.2: Templates for Database-sensitive Prompts

3.2.2 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 3.3 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. Rule **R5** basically encodes this argument as

a rule in Table 3.2. Similarly rule **R6**, which pre-filters tables that are involved in self-joins and have low selectivity filter and/or join predicates, is also used to guide a rewrite. 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.

```
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

```
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 3.3: Example Queries illustrating Rule 5

The input prompt for these rules, as shown in Figure 3.2(b), is modified to include the following:

1. Estimated selectivities of columns appearing in the `WHERE` and `JOIN` clauses – these values are obtained via calls to the cardinality estimation modules of the query optimizer.

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 3.4. We observe that CPRs are now obtained for **9** of the ten micro-benchmark queries. Moreover, the resulting CSGM increases to **10.57**, quite close to the human target of 11.84.

Prompt	**# CPR**	**CSGM**
Basic Prompts $\bigcup$ {R1, ..., R6}	9	10.57

Table 3.4: Performance of Metadata-infused Prompts on micro-benchmark.

We note in closing that rules R1 through R6 not only add queries to the productive category, but also deliver greater improvement for those already deemed to be productive via the standard prompts of Section 3.1. These performance gains can further be improved using MCTS technique discussed in technical report [16].

Chapter 4

Implementation Choices

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

4.1 Regression Detection Mechanism

In addition to refining the rules, we explored two strategies for proactively detecting runtime regressions for `LITHE`'s rewrite.

(a) Identifying robust plan: Query optimizers often make mistakes in selectivity estimation at various points within the execution plan due to inadequate modeling. These inaccuracies frequently lead to suboptimal plans. In our experiments, we observed that the optimizer tends to favor nested loop joins—even in scenarios where a hash join would be more suitable—because of flawed selectivity estimations.

Such discrepancies are common across query optimizers. To ensure that `LITHE` 's rewritten queries do not suffer from runtime regressions, we draw inspiration from the well-established SEER algorithm [21], which offers a analytical framework for identifying robust execution plans.

In our approach, we consider a user query Q^U and its rewritten counterpart Q^T. Let P^U and P^T be the plans corresponding to the original query, Q^U, and the recommended rewrite, Q^T, respectively. We construct *parametrized* versions of these queries, where the constants in the filter predicates are replaced by variables. Then, by assigning appropriate values to these parameters, we construct queries that are located at the *corners* of the selectivity space. The plans P^U and P^T are forced at each of these corner locations. Figure 4.1 illustrates this process for original query in a 2-dimensional selectivity space, where the x-axis represents the selectivity of the predicate `Age > m` and the y-axis represents the selectivity of `Salary > n`. A selectivity of 0% indicates a filter that excludes all tuples during execution, while 100% represents no filtering.

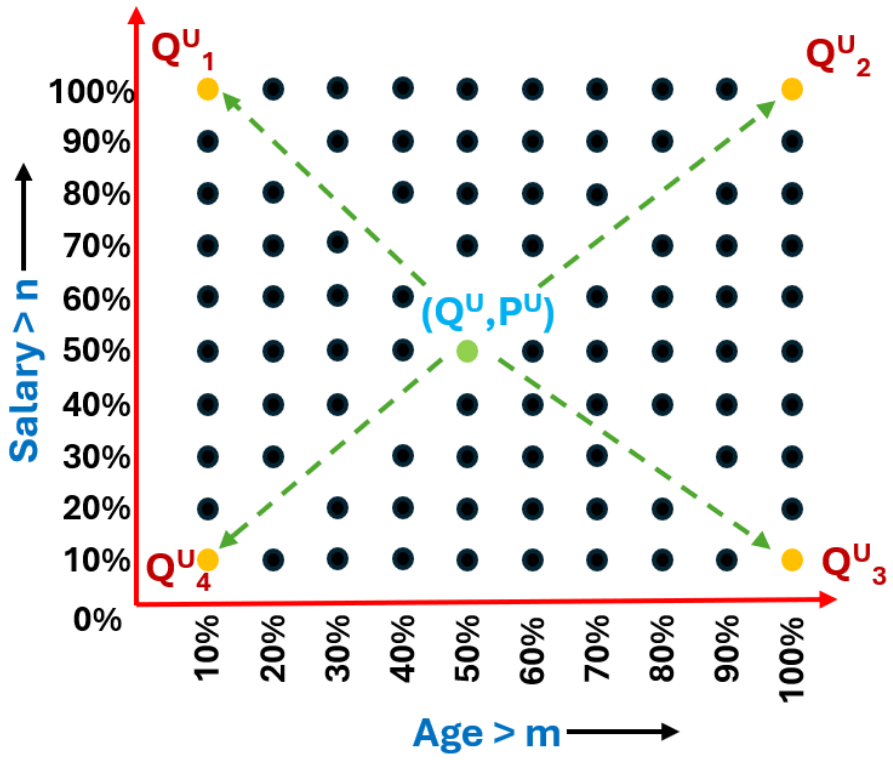


Figure 4.1: Robust plan identification

Here, Q^U is the original query, and Q_1^U, Q_2^U, Q_3^U, and Q_4^U represent the four corner points in the selectivity space. We force the original query's plan P^U at each of these corner points, and apply the same procedure for the rewritten query Q^T. The underlying intuition is that if the rewritten query's plan consistently outperforms the original at extreme points in the selectivity space, it is likely to deliver robust performance over broader selectivity space. Hence, if the rewrite's cost is lower than the original at *all* of them, the rewrite is deemed to be robust.

The foreign plan forcing feature is supported by industrial-strength optimizers, including DB2(Optimization Profile)[6], SQL Server(XML Plan)[1] and Sybase(Abstract Plan)[7].

(b) Runtime Heuristics: Although the SEER algorithm has proven highly effective in identifying robust execution plans, its applicability is currently limited by the lack of plan forcing support across all database engines. In such environments, we must design alternative heuristics to detect potential regressions.

Up to this point, our decision to accept or reject a query rewrite has been based solely on the optimizer's estimated cost. However, a promising extension is to incorporate actual runtime behavior into this decision-making process. Executing queries on the full-scale database to gather such data is often prohibitively expensive—particularly when dealing with database at

the terabyte scale. To address this, we propose leveraging sampled versions of the database.

Notably, we already run queries on the sampled database to verify semantic equivalence between the original and rewritten versions. During this process, we can also collect runtime measurements for both versions. These measurements serve as lightweight runtime heuristics. By comparing the execution times across multiple runs on the sampled database, we can estimate the robustness of a rewrite. If the rewritten query consistently performs better—i.e., shows lower execution times in all sampled runs—it is considered a strong candidate for being robust in the full-scale environment as well.

Section 5.5.1 highlights, the effectiveness of our regression identification mechanism in accurately flagging significant runtime regressions.

4.2 Query Equivalence Testing

Maintaining query equivalence is a fundamental requirement in any query rewriting framework. Ensuring equivalence is relatively straightforward when using rule-based optimizers like Calcite[11], where rewrites are grounded in formally verified transformations. However, with LLM-generated rewrites, there is no such guarantee—the rewritten query may not always be semantically equivalent to the original.

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

4.2.1 Result Equivalence via Sampling

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.

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:

1. 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.

2. It is possible for two different queries to return same non-empty 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.

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

1. We use *correlated sampling* [47] 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. 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. 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.

To address the second problem (false positives), following approach is taken:

1. 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.
2. Injecting query-specific synthetic tuples in the sample, similar to the XData mutant-killing tool [35], to cover predicate boundary conditions (e.g., inserting tuples with values exactly at the boundary, such as salary = 50000, to distinguish between salary > 50000 and salary >= 50000).

Although testing on carefully curated samples proved highly effective in our experiments—yielding no false positives—there is still no absolute guarantee. Therefore, we introduce logic-based tools in the second stage for additional verification.

4.2.2 Logic-based Equivalence

Although verifying equivalence between arbitrary SQL queries is known to be NP-complete[8], several logic-based tools—such as Cosette[13], SQLSolver[19], VeriEQL[22], and QED[41]—have been developed to prove equivalence over restricted classes of queries. The advantage of such a logic-based approach is that it is definitive in outcome. In our evaluation, we explored whether combining multiple tools could increase our coverage of verified rewrites.

We excluded Cosette from our experiments, as prior studies have shown it to be less effective than more recent tools. VeriEQL, while promising, is restricted to proving bounded equivalence—i.e., equivalence only under a limited number of tuples per table—which does not meet our requirements. This narrowed our choice of logic-based tools to SQLSolver and QED, which together cover a broader range of queries compared to the other tools considered. The

advantage of such a logic-based approach is that it is definitive in outcome. Note that this rewrite has already passed the sampling-based tests described above.

To broaden our verification coverage, we also evaluated the behavioral equivalence tool UNMASQUE [24]. However, it failed to verify any additional rewrites. Given its low success rate and high computational overhead, we chose not to integrate UNMASQUE directly into the `LITHE` pipeline.

Result Equivalence on the Entire Database. 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.

Chapter 5

Experimental Evaluation

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 in computational and financial terms. We finally discuss the impact of alternative platforms wrt database engine, database schema 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 1.1. Specifically, the SOTA collection consists of the following approaches:

1. Baseline LLM prompt [30]: This is Prompt 1 from Section 3.1.
2. Learned Rewrite [49], a purely rule-based rewriter.
3. LLM-R^2 [29], an LLM-guided rule-based rewriter.
4. GenRewrite [30], 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 occasionally generate rewrites that are expected by the optimizer's costing module to regress the performance. For safety, we immediately discard such rewrites, similar to LITHE.

Query Set. Our evaluation includes diverse set of industry standard synthetic benchmarks i.e TPC-DS [14], DSB [17], ARCHER [48], JOB [25] as well as a real world benchmarks i.e, StackOverflow [31].

Testbed. The majority of our experiments were 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 Sections 5.5.1. Note that all experiments were conducted in a cold cache environment.

Metrics. For each rewrite technique, we identified the number of queries for which a CPR (cost productive rewrite with > 1.5 speedup) could be constructed. Subsequently, we computed the CSGM (Cost Speedup Geometric Mean) and TSGM (Time Speedup Geometric Mean) performance obtained by each technique over the set of all CPRs (i.e. CPRs arising from either `LITHE` or `SOTA`).

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

5.1 Overall Benchmarks Result

Table 5.1 shows the speedup achieved by `LITHE` and `SOTA` on the CPR queries across different benchmarks. Impressively, `LITHE` achieves CPR for significant number of queries in both TPC-DS (26 queries) and DSB (9 queries). A large number of these rewrites are *highly productive*, with CSGM values exceeding 11.5 for TPC-DS and 7.7 for DSB. Furthermore, `LITHE` significantly outperforms `SOTA` in terms of CPR coverage, with a factor of 3 for DSB and 2 for TPC-DS.

Benchmark	**CPR**		**CSGM**	
(CPR)	**LITHE**	**SOTA**	**LITHE**	**SOTA**
TPC-DS (27)	26	13	11.5	6.1
DSB (9)	9	3	7.7	1.7
ARCHER (22)	22	19	2.1	1.9
JOB (4)	4	2	1.9	1.4
StackOverflow (2)	2	1	8.7	7.5

Table 5.1: Comparing `LITHE` with `SOTA` on CPR queries

Turning our attention to the other benchmarks (ARCHER, JOB, StackOverflow), the number of CPR queries is smaller due to the predominance of flat SPJ formulations in these benchmarks, which limits the scope for productive rewriting at query space. Nevertheless, `LITHE` continues to achieve CPR even in constraint settings, whereas `SOTA` misses quite a few opportunities. Further, the CSGM of `LITHE` is visibly better than `SOTA`.

```
SELECT amc/pmc AS am_pm_ratio
FROM (
  SELECT COUNT(*) AS amc
  FROM web_sales, household_demographics, time_dim td, web_page
  WHERE ws_sold_time_sk = td.t_time_sk
      AND ws_ship_hdemo_sk = house_demographics.hd_demo_sk
      AND ws_web_page_sk = web_page.wp_web_page_sk
      AND td.t_hour BETWEEN 8 AND 8+1
      AND house_demographics.hd_dep_count = 6
      AND web_page.wp_char_count BETWEEN 5000 AND 5200
    )AS at,
  (SELECT COUNT(*) AS pmc
  FROM web_sales, household_demographics, time_dim td, web_page
  WHERE ws_sold_time_sk = td.t_time_sk
      AND ws_ship_hdemo_sk = house_demographics.hd_demo_sk
      AND ws_web_page_sk = web_page.wp_web_page_sk
      AND td.t_hour BETWEEN 19 AND 19+1
      AND house.hd_dep_count = 6
      AND web_page.wp_char_count BETWEEN 5000 AND 5200
    )AS pt
ORDER BY am_pm_ratio
LIMIT 100;
```

Figure 5.1: Original TPC-DS Q90

5.2 Query Readability

We now turn our attention to evaluating how query rewrites affect readability—an often overlooked but crucial aspect of query quality. Specifically, we aim to examine whether the commonly held belief that "more efficient queries are also more readable" holds true in the context of rewrites produced by `LITHE`.

To assess this, we adopt a straightforward yet intuitive metric for readability: **the number of joins** present in a query. The rationale is that a higher number of joins generally increases structural complexity, making the query harder to interpret, debug, and maintain. Fewer joins, on the other hand, often lead to more compact and understandable queries, especially for users manually inspecting the SQL.

Table 5.2 reports the average number of joins before and after rewriting for both `SOTA` and

```sql
SELECT SUM(CASE WHEN td.t_hour BETWEEN 8 AND 9 THEN 1 ELSE 0 END)) /
    SUM(CASE WHEN td.t_hour BETWEEN 19 AND 20 THEN 1 ELSE 0 END) AS am_pm_ratio
FROM web_sales, household_demographics, time_dim td, web_page
WHERE ws_sold_time_sk = td.t_time_sk
    AND ws_ship_hdemo_sk = house_demographics.hd_demo_sk
    AND ws_web_page_sk = web_page.wp_web_page_sk
    AND (td.t_hour BETWEEN 8 AND 9 OR td.t_hour BETWEEN 19 AND 20)
    AND house_demographics.hd_dep_count = 6
    AND web_page.wp_char_count BETWEEN 5000 AND 5200
ORDER BY am_pm_ratio
LIMIT 100;
```

Figure 5.2: Rewritten TPC-DS Q90

Benchmark (CPR)	**Average # JOINS**		
	Original	**SOTA**	**LITHE**
TPC-DS (26)	4.8	4.4	4
DSB (9)	4.9	5.3	4.7
ARCHER (22)	1.44	1	0.8
JOB (4)	7	7	7
StackOverflow (2)	8	8	8

Table 5.2: `LITHE` vs `SOTA` readability comparison

`LITHE` across multiple benchmark suites. The results clearly show that `LITHE` consistently reduces the number of joins, most prominently for the TPC-DS, DSB, and ARCHER benchmarks. This indicates a strong alignment between `LITHE` 's optimizations and improved readability. In contrast, `SOTA` sometimes increases the join count—most notably in the DSB benchmark—thus making the rewritten queries harder to comprehend than the original versions.

For the JOB and StackOverflow benchmarks, where the potential for transformation is inherently limited due to simpler query structures, both the original and rewritten versions remained largely the same in terms of join count. This suggests that `LITHE` avoids unnecessary rewrites when they do not offer meaningful benefits.

Importantly, `LITHE` never performs worse than the original or `SOTA`-rewritten queries in terms of readability, always maintaining or improving the metric. A concrete example is illustrated in Figures 5.1 and 5.2, which depict Query 90 before and after rewriting by `LITHE`. In this case, the number of joins was reduced from 6 to 3, resulting in a significantly simpler and more readable query.

In summary, `LITHE` not only delivers strong performance gains but also enhances query clarity—an essential trait for developers and DBAs working with complex analytical SQL.

5.3 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.

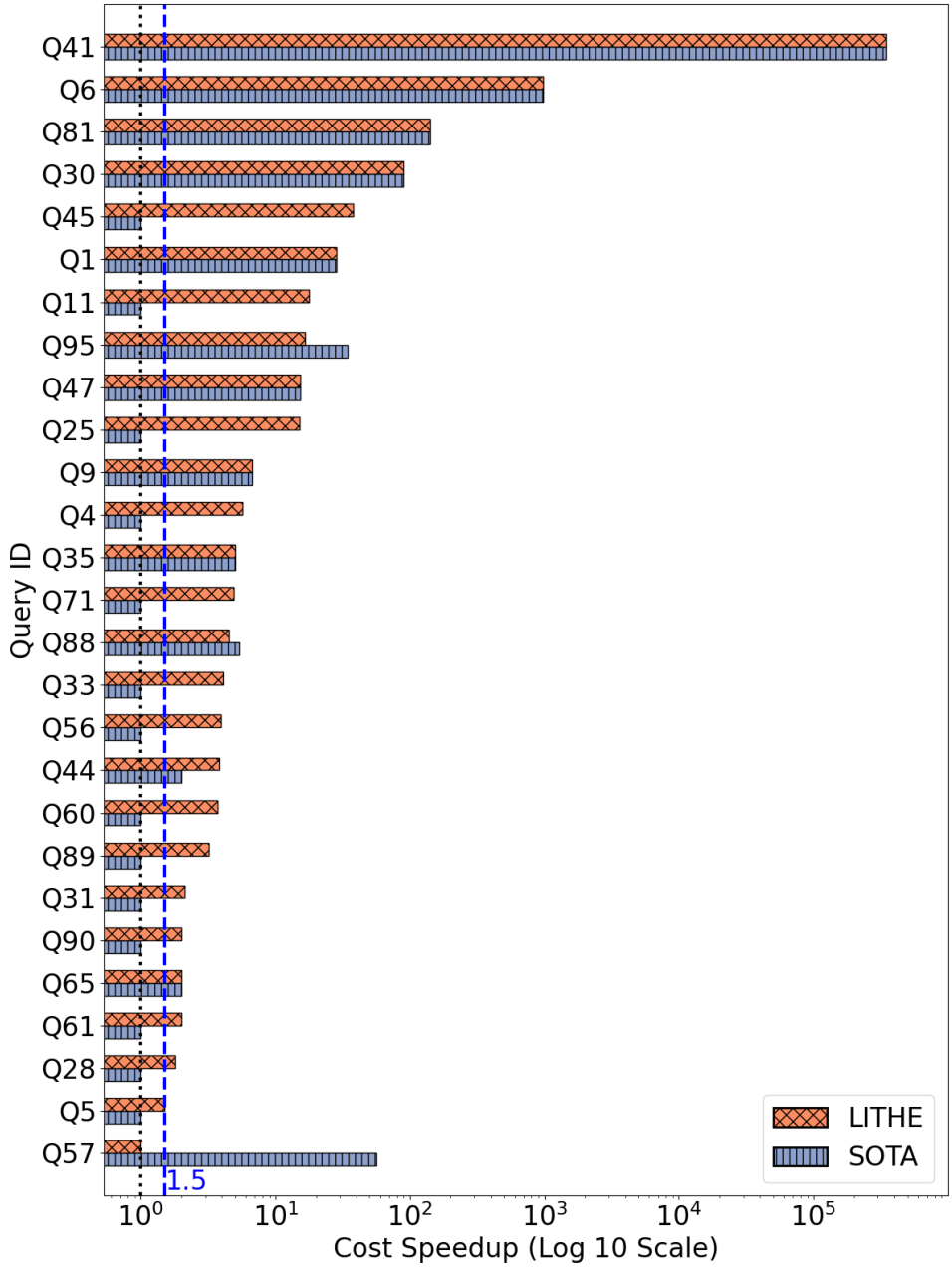


Figure 5.3: Plan Cost Speedups via Rewrites

5.3.1 Estimated Cost

LITHE produces 26 CPRs for the 88 slow TPC-DS queries, resulting in a *highly productive* CSGM of **11.5**, whereas SOTA delivers only 13 CPRs with a CSGM of **6.1**. All but one of the SOTA CPRs also feature in the LITHE CPRs, making total number of CPRs being considered to be 27.

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.

A drill-down into the cost speedup performance at the granularity of individual queries is shown in Figure 5.3, which compares LITHE (orange bars) and SOTA (blue bars) on each of the 27 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 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 rewrites are indeed capable of promising dramatic cost 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.

Second, in most queries, LITHE's cost speedup either exceeds or matches SOTA. In fact, for several queries (e.g. Q45, Q25, Q4), LITHE produces a highly beneficial CPR but SOTA returns the original query. Conversely, the opposite is true for one query (Q57) where SOTA projects a large speedup but LITHE settles for the original query. And in Q88 and Q95, SOTA performs only marginally better.

At this stage, one might expect that adding more rules to LITHE could bring it on par with SOTA for queries like Q57. However, we deliberately include only broad-brush rules in LITHE to ensure generalizability and efficiency. As the following timing section shows, due to this conservative approach LITHE actually outperforms SOTA on these queries (Q57, Q88, Q95) in terms of runtime.

5.3.2 Execution Time

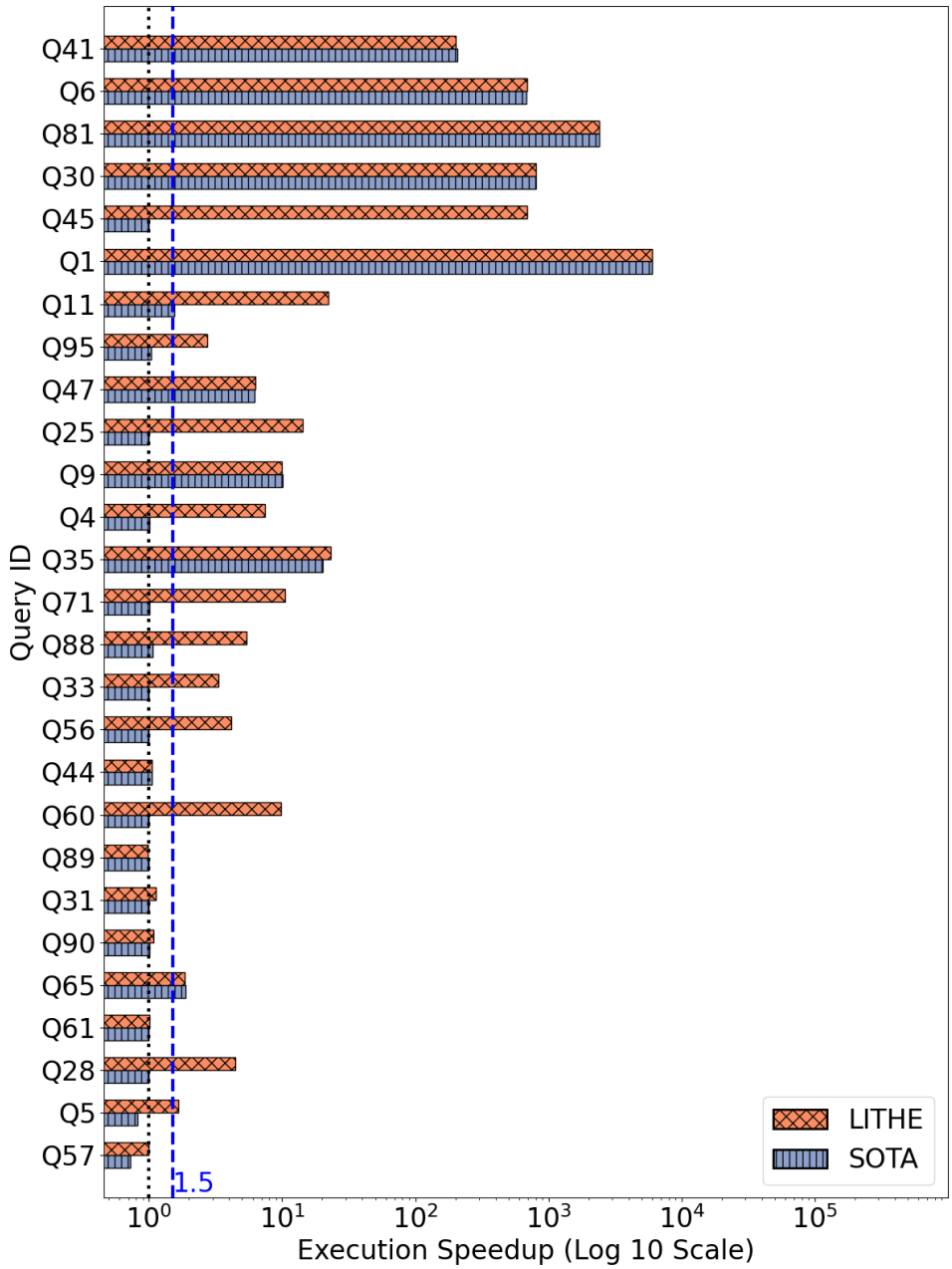


Figure 5.4: Execution Time Speedups via Rewrites

So far, our evaluation has focused on optimizer-estimated execution costs. However, from a user's perspective, what truly matters is the improvement in actual response time—the time it takes for a query to execute and return results.

This makes it essential to evaluate LITHE's performance gains in terms of actual run time. Figure 5.4 shows the runtime speedups (on a $\mathbf{log_{10}}$ scale) achieved by both LITHE and SOTA, offering a direct comparison of real-world performance.

We observe that, LITHE 's performance gains have indeed translated from the estimated cost domain to actual execution time. Gratifyingly, several queries show substantial runtime improvements due to the rewrites—some even achieving order-of-magnitude speedups. For instance, LITHE improves Query 45 by an astonishing factor of 700. Second, in all the cases, LITHE outperforms or matches SOTA, including as mentioned above, the queries where SOTA's optimizer costs were better.

From a modeling standpoint, the gap between optimizer estimates and actual runtimes persists in the rewrite space. For example, Q1's projected 30x speedup jumps to 6000x at runtime, while Q41 drops from 10^5x to 200x. 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 100x.

Encouragingly, LITHE showed no regressions among its CPR rewrites, even if projections weren't always matched. Overall, LITHE achieved a robust TSGM of 13.2, compared to SOTA 's 4.9.

5.4 Rewrite Overheads (Time/Money)

Having established that LITHE can consistently deliver performance-beneficial rewrites, we now shift focus to analyzing the overheads associated with the rewriting process itself.

Benchmark	Avg. Time (min)		Avg. Tokens	
	LITHE	SOTA	LITHE	SOTA
TPC-DS	5	1.7	18427	20076
DSB	9	4.0	15602	15699
ARCHER	2.5	0.6	7284	5465
JOB	5	1.3	13742	13692
StackOverflow	7.3	3.2	20931	12759

Table 5.3: LITHE and SOTA Rewrite time overhead

Table 5.3 presents the average processing time per CPR query across benchmarks. While the rewriting process takes a few minutes on average, such an overhead is generally acceptable in practical deployment scenarios—especially considering that the execution benefits typically far outweigh the compilation overheads.

For instance, with Q11, the original query took nearly *an hours* to complete, whereas the LITHE rewrite executed in just 3 *minutes*, yielding a massive runtime reduction. In this case, even a rewrite time of several minutes becomes negligible in comparison to the performance gain.

The average number of LLM tokens consumed by both LITHE and SOTA during the rewriting process is also reported in Table 5.3. This token count directly correlates with the inference cost, as LLM usage is typically priced per million tokens.

Encouragingly, even though LITHE may involve more extensive prompting due to its deeper rewrite exploration, the overall inference cost per query remains quite modest. Based on current pricing[1], the cost of rewriting a single query amounts to just a few cents.

That said, it is important to note that LITHE 's rewriting process currently incurs higher latency compared to SOTA. This difference primarily stems from LITHE 's design choice to issue multiple prompt-based interactions with the LLM to explore diverse rewrite options. While this enhances the quality and robustness of rewrites, it also increases inference time. However, the rewrite latency can potentially be reduced using techniques such as classifier-based filtering and early pruning, as discussed in the technical report [16].

5.5 Alternative Platforms(Engine/Schema)

5.5.1 Commercial DBMS

A legitimate question could be whether the rewrites made amends for the PostgreSQL optimizer but may fail to be useful in highly-engineered database engines. To evaluate this issue, we performed TPC-DS rewrites on a pair of popular commercial DBMS, OptA and OptB.

	# CPR	
	OptA	OptB
LITHE	12	9
SOTA	3	5

Table 5.4: Rewrite Performance (# CPRs) on Commercial Database

[1]As of this writing, GPT-based inference costs approximately USD 2.50 per million tokens.

	CSGM		TSGM	
	OptA	OptB	OptA	OptB
LITHE	3.6	4.1	2.1	1.9
SOTA	1.5	1.3	1.4	1.2

Table 5.5: Rewrite Performance (CSGM and TSGM) on Commercial Database

The performance of LITHE and SOTA on these two systems is shown in Table 5.4 and 5.5, with LITHE continuing to do better than SOTA. Despite the apparent lack of optimization headroom, LITHE still produces 12 and 9 CPRs resulting in a healthy CSGM of **3.6** and **4.1**, respectively. Further, the TSGM provided by these rewrites are a useful **2.1** and **1.9**, respectively.

Interestingly, although we did not observe any regressions with PostgreSQL, a few did surface in the commercial systems. Nevertheless, our regression identification mechanisms effectively caught these brittle rewrites. As a case in point, a promising rewrite, as estimated by the optimizer, for Q23, actually takes **37 minutes** to complete as compared to **18 minutes** for the original query – this *doubling slowdown* was successfully flagged by the SEER heuristic, and the rewrite was abandoned.

We also conducted a preliminary study of LITHE on an internal real-world benchmark with more realistic data and query characteristics than TPC-DS. Even on this benchmark, LITHE produced 8 CPRs (out of 45 queries tested) with CSGM of 2 and TSGM of 3.3.

The above results suggest LITHE has a useful role to play in industrial environments. 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.

5.5.2 Masked Database

An interesting question that arises is whether the performance gains observed so far might simply be a result of GPT-4o having been extensively trained on the TPC-DS benchmark, which is widely available in the public domain.

To examine this possibility, we constructed a masked version of the TPC-DS database schema, in which all table and column names were replaced with meaningless identifiers—thereby eliminating any semantic cues that might aid the model in understanding the underlying data or intent of the query.

```
SELECT dt.d_year, item.i_brand_id AS brand_id, item.i_brand brand,
    SUM(ss_sales_price) AS sum_agg
FROM date_dim dt, store_sales, item
WHERE dt.d_date_sk = store_sales.ss_sold_date_sk
  AND store_sales.ss_item_sk = item.i_item_sk
  AND item.i_manufact_id = 816 AND dt.d_moy = 11
GROUP BY dt.d_year, item.i_brand, item.i_brand_id
ORDER BY dt.d_year, sum_agg DESC, brand_id;
```

Figure 5.5: Original Query on TPC-DS schema

Figure 5.5 illustrates a representative query using the standard TPC-DS schema, where the table and column names carry clear and descriptive semantics (e.g., catalog_sales, item, sale_price). In contrast, Figure 5.6 shows the same query expressed using the masked schema. As evident, the obfuscated version is based on a randomized schema that offers no intuitive insight into the query's purpose.

```
SELECT dt.tx4_7, tx10.tx10_8 AS brand_id, tx10.tx10_9 AS brand,
    SUM(tx25_14) AS sum_agg
FROM tx4 dt, tx25, tx10
WHERE dt.tx4_1 = tx25.tx25_1
  AND tx25.tx25_3 = tx10.tx10_1
  AND tx10.tx10_14 = 816 AND dt.tx4_9 = 11
GROUP BY dt.tx4_7, tx10.tx10_9, tx10.tx10_8
ORDER BY dt.tx4_7, sum_agg DESC, brand_id;
```

Figure 5.6: Obfuscated Query on Masked schema

We then constructed rewrites for the CPR queries (after syntactic changes to reflect the new masked schema) on this version.

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

Table 5.6: Rewrite Performance (# CPR) on Masked Database.

The results are summarized in Table 5.6 and 5.7, and we observe that the performance profiles for both LITHE and SOTA exhibit only a marginal decline under the masked schema. This highlights the robustness and generalizability of both rewriting approaches. These findings demonstrate that, irrespective of their prior training exposure, LLMs can serve as practical and effective tools for query rewriting—even in environments with obfuscated or non-descriptive database schemas.

Approach	**CSGM**		**TSGM**	
	TPC-DS	**Masked**	**TPC-DS**	**Masked**
LITHE	11.5	10.5	13.2	11.8
SOTA	6.1	5.3	4.9	4.2

Table 5.7: Rewrite Performance (CSGM and TSGM) on Masked Database.

Chapter 6

Conclusions

Based on our study, we now present a few observations with implications for the future design and deployment of rewriting tools.

6.1 Rewrite Space Coverage by LLMs

Given the decades-long body of research dedicated to database query optimization, our initial expectation was that there would be limited room for further performance enhancements through query rewriting. 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 [9]. This combination of structural rewrites and plan-space guidance offers a powerful framework for realizing performance gains beyond what traditional systems can achieve on their own.

6.2 Rewrite Migration to Optimizer

The above demonstrated the potent exploratory power of LLMs. But from an overheads perspective, such rewrites should ideally be within the optimizer's native search space rather than recommended from outside. Therefore, it would be a useful exercise to try and distill fresh optimization rules from these instances, leveraging the extensibility features of contemporary optimizers [18] to facilitate their incorporation in existing systems.

On the flip side, there appears to be an "impedance mismatch" against such integration for certain classes of rewrites. For example, consider the TPC-DS Q90 rewrite in Figure 5.2. The

original query individually computed AM (morning) sales and PM (evening) sales, which were then used to compute the AM to PM ratio. The rewrite, however, extracted all relevant rows in one shot and computed the ratio using CASE statements – encoding such transformations as generic rules in the optimizer appears challenging, given the combinatorial ways in which such transformations can occur.

Therefore, a fruitful area of future research could be achieving a middle-ground between the disparate world-views of LLMs and traditional optimizers.

6.3 Revisiting Optimizer Plan Costing

As highlighted in Section 5.3.2, there were instances of substantive differences between the promised speedup and that delivered at run-time. In fact, to the extent that speedups could even turn out to be regressions! This is due to the brittleness of optimizer plan costing in the new spaces explored by the LLM. During our `LITHE` design process, the prefiltering in Rule R6 (Table 3.2) had initially *not* been restricted to self-joins. It resulted in the number of CPRs (on PostgreSQL) being as high as 65, with an astonishing CSGM of 30.6! However, upon execution, most of these rewrites turned out to be regressions, which led to our inclusion of the restriction. The old rule R6 is shown in Figure 6.1.

	Selectivity Guided Rules
Old 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.
New R6	Pre-filter tables involved in self-joins and with low selectivities on their filter and/or join predicates. Remove any redundant filters from the main query. Do not create explicit join statements.

Table 6.1: Modified rule R6

But note that we are incorporating guardrails to flag such cases, rather than fixing the plan costing module, which is the principled solution. In sum, while plan cost modeling has been a long-standing area of research, there is now even more reason given the new rewrite spaces to study this topic further – for instance, the operator cost model could be extended using calibration techniques similar to those advocated in [44], while the operator cardinality model could be improved with attention-based techniques [28].

6.4 Scope of Semantic Equivalence Tools

As seen in the experiments section, logic-based query equivalence testing covers industrial-strength queries only to a limited extent. On the other hand, while it is highly likely that the statistics-verified rewrites are valid, it still requires the DBA to make a final call on the correctness. This limitation restricts the use of `LITHE` in a fully automated scenario, i.e., as a direct preprocessor to the query engine. Therefore, a key challenge is to improve logic-based coverage.

6.5 Road Ahead

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. To address discrepancies between the optimizer's cost model and real execution behavior, we also implemented an efficient mechanism for regression detection. 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. 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.

Appendix

Examples used in Prompts for Rules 1–6

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 tables that are involved in self-joins and have low selectivities on their filter and/or join predicates. Remove any redundant filters from the main query. Do not create explicit join statements.

Original Query

```
with total_price_cte as (
    select item.id, colour.colorcode, sum(item.price) total_price
    from item, color
    where item.colorcode = colour.colorcode
    group by item.id, colour.colorcode
)
select t_sec.id, t_first.colorcode
from total_price_cte t_first, total_price_cte t_sec
where t_sec.id = t_first.id
    and t_first.colorcode = 'R'
    and t_sec.colorcode = 'R'
    and t_first.total_price > 0
order by t_sec.id
limit 100;
```

Statistics

```
Selectivity of different predicates is given below :
( 1 ) colour.colorcode = 'R' :: 0.01
```

Rewritten Query

```
with total_price_cte as (
    select item.id, colour.colorcode, sum(item.price) total_price
    from item, color
    where item.colorcode = colour.colorcode
    group by item.id, colour.colorcode
),
filtered_total_price_cte as (
    select * from total_price_cte
```

```
    where colorcode = 'R'
)
select t_sec.id, t_first.colorcode
from filtered_total_price_cte t_first, filtered_total_price_cte t_sec
where t_sec.id = t_first.id
    and t_first.total_price > 0
order by t_sec.id
limit 100;
```

Bibliography

[1] Sqlserver xml plan forcing, 2012. URL http://msdn2.microsoft.com/en-us/library/ms189298.aspx. 18

[2] Postgresql release 16, 2023. URL www.postgresql.org/docs/16/release-16.html. 2

[3] 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. 8

[4] Defog sqlcoder, 2024. URL github.com/defog-ai/sqlcoder. 3

[5] Defog sql-eval, 2024. URL https://github.com/defog-ai/sql-eval. 3

[6] Db2 optimization profile, 2025. URL https://www.ibm.com/docs/en/db2/11.5.x?topic=guidelines-optimization-profiles. 18

[7] Sybase abstract plan, 2025. URL https://help.sap.com/docs/SAP_ASE/2293f21ce44949b7a4d0ea8a052d178d/a9811637bc2b101497288c927e673849.html?&version=16.0.1.0. 18

[8] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. Equivalences among relational expressions. *SIAM J. Comput.*, 8(2):218–246, 1979. doi: 10.1137/0208017. URL https://doi.org/10.1137/0208017. 20

[9] 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. 4, 34

[10] 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. 2, 3

[11] 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, 19

[12] 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. 2, 3

[13] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. Cosette: An automated prover for SQL. In *8th Biennial Conference on Innovative Data Systems Research, CIDR 2017, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*. www.cidrdb.org, 2017. URL http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf. 20

[14] The Transaction Processing Performance Council. TPC Benchmark™ DS (tpc-ds). In *TPC Benchmark DS (Decision Support)*, 2006. URL www.tpc.org/tpcds/. 5, 22

[15] 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. 7

[16] 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. iii, 8, 16, 30

[17] 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. 22

[18] Bailu Ding, Vivek Narasayya, and Surajit Chaudhuri. Extensible query optimizers in practice. *Foundations and Trends® in Databases*, 14(3-4):186–402, 2024. ISSN 1931-7883. doi: 10.1561/1900000077. URL http://dx.doi.org/10.1561/1900000077. 34

[19] 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. 20

[20] 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

[21] Harish Doraiswamy, Pooja N. Darera, and Jayant R. Haritsa. Identifying robust plans through plan diagram reduction. *Proc. VLDB Endow.*, 1(1):1124–1140, 2008. doi: 10.14778/1453856.1453976. URL http://www.vldb.org/pvldb/vol1/1453976.pdf. 17

[22] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. Verieql: Bounded equivalence verification for complex SQL queries with integrity constraints. *CoRR*, abs/2403.03193, 2024. doi: 10.48550/ARXIV.2403.03193. URL https://doi.org/10.48550/arXiv.2403.03193. 20

[23] Sylvio Barbon Junior, Paolo Ceravolo, Sven Groppe, Mustafa Jarrar, Samira Maghool, Florence Sèdes, Soror Sahri, and Maurice van Keulen. Are large language models the new interface for data pipelines? In Philippe Cudré-Mauroux, Andrea Kö, and Robert Wrembel, editors, *Proceedings of the International Workshop on Big Data in Emergent Distributed Environments, BiDEDE 2024, Santiago, Chile, June 9-15, 2024*, pages 6:1–6:6. ACM, 2024. doi: 10.1145/3663741.3664785. URL https://doi.org/10.1145/3663741.3664785. 4

[24] Kapil Khurana and Jayant R. Haritsa. Unmasque: a hidden sql query extractor. *Proc. VLDB Endow.*, 13(12):2809–2812, aug 2020. ISSN 2150-8097. doi: 10.14778/3415478.3415481. URL doi.org/10.14778/3415478.3415481. 21

[25] 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. 22

[26] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. LLM for data management. *Proc. VLDB Endow.*, 17(12):4213–4216, 2024. doi: 10.14778/3685800.3685838. URL https://www.vldb.org/pvldb/vol17/p4213-li.pdf. 4

[27] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls. 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/83fc8fab1710363050bbd1d4b8cc0021-Abstract-Datasets_and_Benchmarks.html. 3

[28] Pengfei Li, Wenqing Wei, Rong Zhu, Bolin Ding, Jingren Zhou, and Hua Lu. Alece: An attention-based learned cardinality estimator for spj queries on dynamic workloads. *Proc. VLDB Endow.*, 17(2):197–210, October 2023. ISSN 2150-8097. doi: 10.14778/3626292.3626302. URL https://doi.org/10.14778/3626292.3626302. 35

[29] 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, 5, 22

[30] 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, 5, 10, 22

[31] Ryan Marcus. Stack dataset, 2021. URL rmarcus.info/stack.html. 22

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

[33] 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. 2

[34] 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. 3

[35] Shetal Shah, S. Sudarshan, Suhas Kajbaje, Sandeep Patidar, Bhanu Pratap Gupta, and Devang Vira. Generating test data for killing SQL mutants: A constraint-based approach. In Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan, editors, *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, pages 1175–1186. IEEE Computer Society, 2011. doi: 10.1109/ICDE.2011.5767876. URL https://doi.org/10.1109/ICDE.2011.5767876. 20

[36] 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

[37] 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. 3

[38] Jie Tan, Kangfei Zhao, Rui Li, Jeffrey Xu Yu, Chengzhi Piao, Hong Cheng, Helen Meng, Deli Zhao, and Yu Rong. Can large language models be query optimizer for relational databases? *CoRR*, abs/2502.05562, 2025. doi: 10.48550/ARXIV.2502.05562. URL https://doi.org/10.48550/arXiv.2502.05562. 4

[39] Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprintarxiv:2312.11805*, 2023. 2

[40] Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Qian-Wen Zhang, Zhao Yan, and Zhoujun Li. MAC-SQL: A multi-agent collaborative framework for text-to-sql. *CoRR*, abs/2312.11242, 2023. doi: 10.48550/ARXIV.2312.11242. URL https://doi.org/10.48550/arXiv.2312.11242. 3

[41] 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. 20

[42] Yifan Wang, Haodi Ma, and Daisy Zhe Wang. No more optimization rules: Llm-enabled policy-based multi-modal query optimizer. *CoRR*, abs/2403.13597, 2024. doi: 10.48550/ARXIV.2403.13597. URL https://doi.org/10.48550/arXiv.2403.13597. 4

[43] 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. 2

[44] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1081–1092. IEEE Computer Society, 2013. doi: 10.1109/ICDE.2013.6544899. URL https://doi.org/10.1109/ICDE.2013.6544899. 35

[45] 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. 2, 3

[46] 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. *CoRR*, abs/2312.17449, 2023. doi: 10.48550/ARXIV.2312.17449. URL https://doi.org/10.48550/arXiv.2312.17449. 3

[47] Feng Yu, Wen-Chi Hou, Cheng Luo, Dunren Che, and Mengxia Zhu. CS2: a new database synopsis for query estimation. In Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 469–480. ACM, 2013. doi: 10.1145/2463676.2463701. URL https://doi.org/10.1145/2463676.2463701. 20

[48] 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. 22

[49] 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. 2, 3, 5, 22

[50] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. DB-GPT: large language model meets database. *Data Sci. Eng.*, 9(1):102–111, 2024. doi: 10.1007/S41019-023-00235-6. URL https://doi.org/10.1007/s41019-023-00235-6. 3, 4