GenRewrite: Query Rewriting via Large Language Models

JIE LIU*, University of Michigan, USA
BARZAN MOZAFARI, University of Michigan, USA

Query rewriting is an effective technique for refining poorly written queries before they reach the query optimizer. However, manual rewriting is not scalable, as it is prone to errors and requires deep expertise. Traditional query rewriting algorithms fall short too: rule-based approaches fail to generalize to new query patterns, while synthesis-based methods struggle with complex queries. Fortunately, Large Language Models (LLMs) already possess broad knowledge and advanced reasoning capabilities, making them a promising solution for tackling these longstanding challenges.

In this paper, we present GenRewrite, the first holistic system that leverages LLMs for query rewriting beyond traditional rules. We introduce the notion of Natural Language Rewrite Rules (NLR2s), which serve as hints for the LLM while also a means of knowledge transfer from rewriting one query to another, allowing GenRewrite to become smarter and more effective over time. We present a novel counterexample-guided technique that iteratively corrects the syntactic and semantic errors in the rewritten query, significantly reducing the LLM costs and the manual effort required for verification. Across the standard TPC-DS [7] and JOB [23] benchmarks and their SQLStorm-generated variants [36], GenRewrite consistently optimizes more queries at every speedup threshold than all baselines. At the ≥2x threshold on TPC-DS, GenRewrite improves 25 queries— 1.35x more than LLM-driven baselines and 2.6x more than LLM-enhanced rule-based baselines—and the gap widens further on TPC-DS (SQLStorm); on JOB and its SQLStorm variant, where queries are simpler, absolute gains are smaller but GenRewrite still leads by a notable margin.

CCS Concepts: • **Information systems** → **Query optimization**; • **Computing methodologies** → *Artificial intelligence.*

Additional Key Words and Phrases: Query Rewriting, Natural Language Rewrite Rules, SQL Optimization

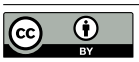
ACM Reference Format:
Jie Liu and Barzan Mozafari. 2026. GenRewrite: Query Rewriting via Large Language Models. *Proc. ACM Manag. Data* 4, 1 (SIGMOD), Article 70 (February 2026), 26 pages. https://doi.org/10.1145/3786684

1 Introduction

Inefficient queries are a significant problem for nearly any organization relying on a database system. These poorly written queries—whether authored by inexperienced users or auto-generated by business intelligence (BI) tools—are a major cause of slow performance and, in cloud databases like Snowflake [12] or BigQuery [16], can lead to excessive costs. As a result, SQL-to-SQL query rewriting is a crucial step in optimizing performance and reducing expenses, as the likelihood of the query optimizer finding an efficient query plan is significantly diminished if it is given a

*Corresponding author.

Authors' Contact Information: Jie Liu, University of Michigan, Ann Arbor, Michigan, USA, jiezzliu@umich.edu; Barzan Mozafari, University of Michigan, Ann Arbor, Michigan, USA, mozafari@umich.edu.



ACM 2836-6573/2026/2-ART70
https://doi.org/10.1145/3786684

poorly-written query to begin with. However, whether performed manually or automatically, query rewriting has remained a challenging task.

Challenges of manual query rewriting. The poorly-written queries that have the biggest impact on the overall cost and performance tend to be quite complex too. For example, it is not uncommon for queries auto generated by Looker [3] to span 100s of lines. Ensuring the rewritten query preserves the semantics of the original requires a deep understanding of the database schema, constraints, and the user intent, and is therefore a time-consuming and error-prone task, even for seasoned database experts. The fact that there are often thousands, if not millions, of such slow queries makes the manual approach a monumental undertaking.

Limitations of traditional automated query rewriting. Despite extensive research, automated query rewriting techniques have long faced fundamental limitations that hinder their practical effectiveness. The most common approach relies on *rewrite rules*––pattern-matching transformations that replace query fragments with optimized equivalents while preserving semantics. Regardless of whether the rules are developed and provided by experts [9, 11, 33] or are inferred automatically [14, 44], a rule-based query rewriter is as effective as the set of rules provided to it. By definition, rule-based query rewriters[1] are fundamentally incapable of optimizing new query patterns that do not match their existing rules [15]. Although synthesis-based query-rewriting does not require prior rules [15], in practice it only handles simple queries—for instance, SlabCity [15] optimizes just 2 out of 22 TPC-H [8] queries and 3 out of 99 TPC-DS [7] queries.

Large language models (LLMs). The shining success of large language models (LLMs) in performing complex and open-ended tasks has created new hopes for solving some of the hardest database problems as well. While researchers have used LLMs for Text-to-SQL [17, 18, 21, 34, 40] and a few other areas, e.g., data cleaning and integration [31, 39], table processing [26, 28] and database diagnosis [48], their potential for query rewriting has remained largely unexplored. In instances where human experts or existing rules cannot rewrite a query, say for complex queries or new query patterns, could LLMs be leveraged to uncover rewrite opportunities due to their broad general knowledge and advanced reasoning capabilities? If so, this could drastically lower the burden on human experts and allow for a much larger set of queries to benefit from rewriting. To the best of our knowledge, this paper is the first[2] to focus on answering this question: how to best leverage LLMs for query rewriting beyond rules and how to address the challenges involved.

Challenges. Using LLMs for query rewriting has challenges:[3]

C1 The naïve approach of simply prompting the LLM to "rewrite the given query into a faster but equivalent form" can rarely handle complex queries: most outputs either fail to parse (syntactic errors) or produce incorrect results (semantic errors) due to SQL's complexity and LLM hallucinations (§ 5.2).

C2 LLMs lack a database cost model and cannot run experiments, so their rewrites are not necessarily faster than the original queries.

C3 While LLM costs for rewriting a recurrent workload can be amortized, excessive invocations can still become expensive.

C4 Hints can help the LLM, but excessive or irrelevant ones can lead to mistakes.

[1]Rule-based query *rewriting* should not be confused with rule-based query *optimization*; the latter is simpler, and thus much more common in practice [19].

[2]See §7 for related work published after the initial publication of our work on ArXiv.

[3]A common misunderstanding is that the time and monetary overhead of invoking an LLM may outweigh the speedup benefits of the rewrite. However, this one-time overhead is less relevant for repetitive workloads, which are the focus of GenRewrite (see **Target workloads**).

C5 Due to their black-box nature, LLM rewrites are harder for humans to understand and verify than traditional rules.

Our approach. To address the above challenges, we present **GenRewrite**, the first holistic system that leverages LLMs for query rewriting. We introduce the notion of a Natural Language Rewrite Rule (NLR2), which is a textual explanation summarizing the rewrite. We use the LLM itself to produce the NLR2s, which are then used for three purposes. The NLR2s 1) serve as hints assisting the LLM to provide better rewrites (C2), requiring fewer follow-up LLM invocations (C4), 2) make the rewrites easier to understand and verify by offering a human-readable explanation (C5), and, most importantly, 3) allow GenRewrite to transfer the knowledge gained from rewriting one query to another and thus become smarter and more effective over time (C1, C2, C3). Since NLR2s are void of confidential information (such as column or schema names), one can even bootstrap GenRewrite's NLR2 repository for a new workload using the repository of a previously trained GenRewrite on another workload.

To avoid redundant rules, which can in turn confuse the LLM, we maintain a utility score for each NLR2 to supply only those rules to the LLM as hints that are most relevant to the query at hand. Finally, to minimize the LLM cost and minimize the human effort needed for manual verification, GenRewrite does not discard incorrect rewrites. Instead, it uses a novel counterexample-guided technique to iteratively correct the syntactic and semantic errors in the rewritten query (C1, C3).

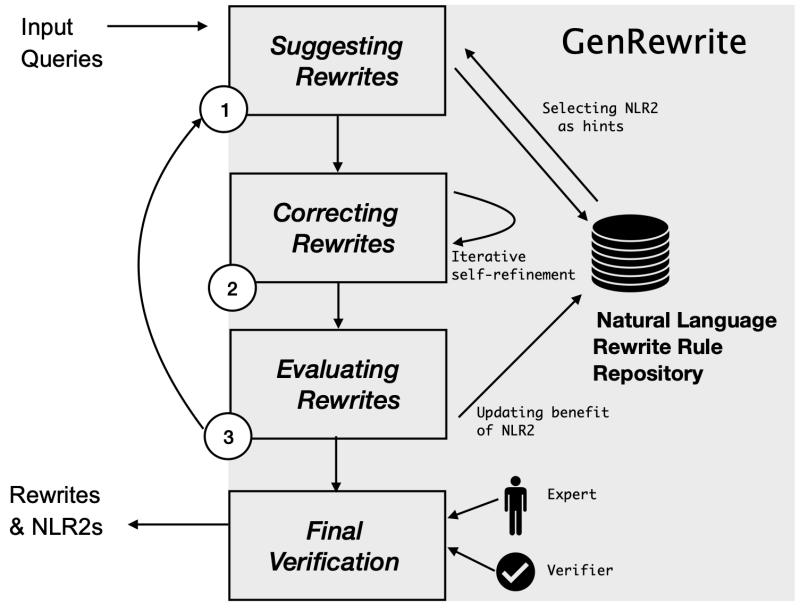


Fig. 1. High-level Workflow of GenRewrite

High-level workflow. GenRewrite takes as input a workload,[4] expressed as a set of input queries Q, and returns $\{(q,q',e) \mid q \in Q_{opt}\}$, where $Q_{opt} \subseteq Q$ is the optimizable subset of the input queries, q' is the rewritten query that is equivalent to q but likely to run faster, and e is a human-readable explanation of the rewrite. Figure 1 shows the overall workflow, which is an iterative process. In each iteration, every query in the workload goes through three phases to find a better rewrite than the one found in previous iterations: ① rewrite suggestion, ② rewrite correction, and ③

[4]A streaming setting is a special case, whereby queries are presented and rewritten one at a time.

rewrite evaluation. Additionally, GenRewrite maintains a Natural Language Rewrite Rule (NLR2) repository for knowledge transfer across queries in the workload. ① selects appropriate NLR2s from the repository to serve as hints for suggesting new rewrites. Any newly discovered NLR2 is also added to the repository. ② iteratively refines rewrites based on the counterexamples provided by the LLM to eliminate semantic and syntactic errors. ③ evaluates the rewrites for equivalence and performance, and updates the benefit score of the used NLR2s accordingly. This process continues until no additional improvement is found and the set of rewrites stabilizes. The rewrites then go through a final verification step ④, where if the formal verifier cannot guarantee equivalence automatically, a human expert is asked to manually inspect and confirm the rewritten query's equivalence before it is used or deployed to production.

Target workloads. Repeatedly invoking an LLM incurs a time and monetary overhead. As mentioned above, when the formal verifier cannot guarantee the equivalence, a human expert needs to inspect the rewrite, which incurs an additional delay. For these reasons, GenRewrite is only focused on the following use cases:

(1) **Recurrent workloads**, where the same query template is executed many times, such as BI (Business Intelligence) dashboards, reporting, ETL, DB-based applications, and dbt [1] jobs. For instance, over 60% of the jobs in Microsoft clusters are recurrent [20]. Here, queries can be rewritten and optimized by GenRewrite once and then reused many times.
(2) **Highly expensive queries** expected to take 10s of minutes, e.g., an adhoc query scanning terabytes of data. Here, even if the query is run once, the additional overhead of leveraging an LLM or a human expert is still well justified.

In other words, GenRewrite is *not* designed for real-time adhoc workloads, where queries are novel **and** relatively light, as the additional overhead of an LLM cannot be amortized.

Contributions. We make the following contributions:

(1) We present one of the first in-depth analyses of LLMs for query rewriting, outlining key challenges and opportunities for future research in this area (§3).
(2) We present GenRewrite, a holistic tool leveraging LLMs for autonomous query rewriting. It addresses the key challenges of using LLMs by introducing (i) the notion of Natural Language Rewrite Rules (NLR2) for finding better rewrites and effective knowledge transfer, (ii) a utility score for NLR2s to minimize LLM confusion caused by irrelevant or too many rules , and (iii) a counterexample-guided iterative correction method. (§4).
(3) We evaluate GenRewrite on the standard TPC-DS and JOB benchmarks and their SQLStorm-generated variants. GenRewrite consistently optimizes more queries at every speedup threshold than all baselines. At the ≥2x threshold on TPC-DS, GenRewrite improves 25 queries— 1.35x more than LLM-driven baselines and 2.6x more than LLM-enhanced rule-based baselines—and the gap widens further on TPC-DS (SQLStorm); on JOB and its SQLStorm variant, where queries are simpler, absolute gains are smaller but GenRewrite still leads. (§5).

2 Motivation and Background

2.1 Limitations of Traditional Query Rewriting

Despite decades of effort by experts to craft extensive rewrite rules, many queries still cannot be optimized by existing rules [15]. In those cases, leveraging general knowledge becomes essential in identifying redundant calculations in the query and exploring alternative ways of achieving the same output. To demonstrate this, we present two examples: one from LeetCode [2], the other from TPC-DS [7], an industry-standard business intelligence benchmark.

```sql
select max(a.salary) as secondHighest
from employee as a, employee as b
where a.salary < b.salary
```

Q1: Cartesian product

```sql
select max(salary) as secondHighest
from employee
where salary < (select max(salary) from
      employee)
```

Q2: with subquery

Fig. 2. Finding the second highest salary

Example 1: LeetCode #176. The goal is to find the second highest salary from the Employee table. Q1 in Figure 2 is the human-written solution (submitted by LeetCode users) while Q2 emerges as a rewrite generated by LLMs. Q2 is approximately 2000x faster than Q1 when executed on a database populated with one million rows. The optimization process involves a deep understanding of the query intent through reasoning: (1) identifying that the goal of Q1 is to find the second-highest salary, (2) realizing that there is an alternative way to achieve the same thing, i.e., first find the maximum salary using a subquery and then compare all salaries with this maximum salary to find the second maximum, and finally (3) recognizing that Q2 achieves performance improvement by reducing the number of comparisons required. No existing rewrite rule explicitly captures this transformation. As a result, this optimization cannot be achieved using traditional rule-based query rewriting techniques, highlighting the advantage of LLM-guided rewriting in discovering non-trivial, high-impact transformations.

```sql
with year_total as (
    select ..., 's' sale_type
    from store_sales, ...
    ...
    union all
    select ..., 'w' sale_type
    from web_sales, ...
    ...
)
select
    s_2.customer_id, ...
from
    year_total s_1,
    year_total s_2,
    year_total w_1,
    year_total w_2
where
    s_2.customer_id = s_1.customer_id
    and s_1.customer_id = w_2.customer_id
    and s_1.customer_id = w_1.customer_id
    and s_1.sale_type = 's'
    and s_1.dyear = 1998
    and w_1.sale_type = 'w'
    and w_1.dyear = 1998
    and s_2.sale_type = 's'
    and s_2.dyear = 1999
    and w_2.sale_type = 'w'
    and w_2.dyear = 1999
    and w_2.y_total / w_1.y_total >
        s_2.y_total / s_1.y_total
```

```sql
with store_sales_total as (
       select ...
       from store_sales, ...
       ...
   ),
   web_sales_total as (
       select ...
       from web_sales, ...
       ...
   )
select
    s_2.customer_id, ...
from
    store_sales_total s_1,
    store_sales_total s_2,
    web_sales_total w_1,
    web_sales_total w_2
where
    s_2.customer_id = s_1.customer_id
    and s_1.customer_id = w_2.customer_id
    and s_1.customer_id = w_1.customer_id
    and s_1.dyear = 1998
    and w_1.dyear = 1998
    and s_2.dyear = 1999
    and w_2.dyear = 1999
    and w_2.y_total / w_1.y_total >
        s_2.y_total / s_1.y_total
```

Listing 1. Left: Original TPC-DS Q11 query. Right: A rewrite of Q11 generated by LLMs. The rewrite is 9x faster, though no existing rewrite rules can achieve this transformation.

Example 2: TPC-DS Q11. In Listing 1 (left), a query adapted from TPC-DS Q11 identifies customers whose web sales growth from 1998 to 1999 outpaces their store sales growth over the same period. It begins by creating a Common Table Expression (CTE) `year_total`, which aggregates annual

store and web sales data for each customer (lines 2–4 and 6–8) via a **`UNION ALL`** operation (line 5). The query then filters this aggregated data by sales type and year to isolate the figures for 1998 and 1999 (lines 21–28). A join on `customer_id` (lines 18–20) allows for a comparison of each customer's sales across different years and channels. In essence, this query uses **`UNION ALL`** to combine multiple data sources with a column to label each source, and then filtering on the label later to separate them. The combining and filtering cancel each other's effect and thus can be removed for better performance and readability. The revised query in Listing 1 (Right) makes two changes: (1) creates separate CTEs for store and web sales data, and (2) removes the sale type filters. This results in a 9x speedup, reducing execution time from 18 minutes to 2 minutes on PostgreSQL with TPC-DS scale factor 10. The performance boost is due not only to avoiding unnecessary computations but also to PostgreSQL's cardinality estimation errors with more predicates.

No existing rewrite rules address this anti-pattern, preventing rule-based query rewriting techniques from discovering this more efficient rewrite. However, a database expert reviewing the example would intuitively reason that the **`UNION ALL`** operation and the subsequent filtering cancel each other out. However, manually scrutinizing every query for new anti-patterns is not feasible. LLMs with advanced reasoning capabilities offer a promising alternative for automatically identifying and exploiting new rewrite opportunities.

2.2 Large Language Models: Background

Recent advancements in deep learning and the availability of large datasets have propelled LLMs into the spotlight. These models exhibit advanced reasoning skills, allowing them to take on open-ended tasks and make logical inferences.

Prompt engineering. The key to leveraging LLMs effectively lies in *prompt engineering*[6]. This involves crafting precise inputs to guide the model's output. By providing natural language instructions, users can guide the model towards the desired outcome. Through iterative refinement, prompt engineering allows for high-quality and relevant responses, giving better control over creativity and consistency. In query rewriting, we concatenate an input query q with a prompt p, denoted as $p||q$, which encodes a comprehensive task description for rewriting. Figure 3 is an example of a basic prompt with only the query and a concise prompt, which we refer to as Baseline LLM.

Tokens and cost. Tokens are the units or building blocks that LLMs use to process language. For example, the string " tokenization" is decomposed as two tokens " token" and "ization", while a short common word like " the" is represented as a single token [5]. The cost of using LLMs is typically determined by the number of tokens in both the input and the output.

3 Lessons Learned from employing LLMs for query rewriting

We conducted a comprehensive experiment to study the applicability of LLMs for rewriting of a variety of queries and improving their performance. Specifically, we carefully reviewed each rewrite suggested by LLMs and evaluated it for correctness and latency. We paid particular attention to discerning which problems could be effectively addressed and which could not, along with identifying conditions under which the latter could be transformed into the former. Our study not only provided valuable insight into the capabilities of LLMs and the challenges associated with using it for query rewriting but also contributed to the development and refinement of GenRewrite that is presented in §4. Below we summarize the main lessons that have informed GenRewrite's design decisions.

Lesson 1: LLMs are effective on simple workloads but struggle with complex rewrites. While LLMs can identify effective rewrites for simpler queries using only basic prompts, their

```
[input query]
Rewrite this query to improve performance.
```

Figure 3. The baseline approach for LLM-based query rewriting (Baseline LLM)

ability to recognize and apply meaningful transformations diminishes as query complexity increases. Although it remains possible for LLMs to discover rewrite opportunities, the decreasing likelihood necessitates more iterations of LLM runs, resulting in significantly higher costs. In §2, we demonstrated how Q11 in TPC-DS workload could be optimized into a more efficient form by eliminating the unnecessary **`UNION ALL`** . Similar inefficiencies exist in other TPC-DS queries, such as Q4 and Q74. Among them, Q4 stands out as the most complex: it has the highest token count (1,170 vs. 723 in Q11 and 591 in Q74), the greatest number of **`UNION ALL`** segments (3 vs. 2 for the others), and the most joins (5 vs. 3 for the others). Despite ten attempts using the prompt shown in Figure 3 with GPT-4, none of the rewrites for Q4 suggested removing the **`UNION ALL`** , a transformation that would have significantly reduced execution time. In contrast, 4 out of 10 attempts for Q11 and 6 out of 10 for Q74 moved in the correct rewriting direction. Note that a correct rewriting direction, such as the suggestion to eliminate unnecessary **`UNION ALL`** in Q74's rewrites, does not guarantee an error-free or equivalent rewrite to the original query. The inherent complexity of such queries makes it much more difficult for LLMs to generate fully correct rewrites, even when the high-level transformation logic is sound.

Lesson 2: LLMs can optimize queries that they previously struggled to optimize when provided with suitable rewrite hints in the prompt. As zero-shot prompting is not universally effective for query rewriting task, it becomes essential to incorporate precise guidance within the prompts. Drawing inspiration from the few-shot prompting used in Text-to-SQL tasks [40], one approach is to provide LLMs with examples that include both the input query and its optimized version. Yet, an alternative strategy —using concise, human-readable rewrite hints— also holds promise. For example, one may intuitively recognize that both queries can benefit from "avoid the unnecessary **`UNION ALL`** ". In fact, integrating this insight as a hint in the prompt consistently leads LLMs to suggest removing **`UNION ALL`** across different attempts. This latter method offers two benefits over the few-shot prompting approach: (1) it requires fewer tokens, thereby reducing costs and accelerating the rewrite generation process, and (2) the hints are easily interpretable by humans, facilitating quicker debugging. However, due to the time constraints of human experts, manually analyzing each query and suggesting beneficial rewrite hints is impractical. Therefore, there is a need for an automated method to identify rewrite hints tailored for each query.

4 Our Approach

In this section, we describe our proposed algorithm, which incorporates the lessons presented in §3.

The iterative rewriting pipeline in GenRewrite consists of three core components: (1) suggesting high-quality candidate rewrites, (2) iteratively correcting these rewrites for equivalence, and (3) evaluating their effectiveness. This pipeline closely integrates with GenRewrite's **Natural Language Rewrite Rule (NLR2) Repository**, enabling improved knowledge transfer and more fine-grained control over the query rewriting process.

GenRewrite's top-level algorithm is presented in Algorithm 1. Initially, the algorithm takes a set of queries Q as input for optimization. NLR2 Repository $\mathcal{R}$ may either be initialized as empty or pre-populated with an offline-constructed repository $\mathcal{R}_{pre}$. At each iteration, GenRewrite explores the NLR2 repository to identify the most relevant NLR2s for the target query, integrating them into the rewriting prompt. If no suitable NLR2s are found,GenRewrite defaults to using a basic prompt.

[Input query]
Rewrite this query to improve performance. Only use this rule when rewriting:
[a selected NLR2]

Figure 4. A prompt incorporating a selected NLR2.

The LLM suggests candidate rewrites, accompanied by explanations describing the applied NLR2s and the conditions under which they are applicable. Through iterative correction and feedback mechanisms, GenRewrite progressively converges towards generating more accurate rewrites, thereby largely mitigating the issue of inequivalence. Semantic correction feedback is obtained from language models, while syntax correction relies on feedback from the database. Subsequently, our tool evaluates rewrites for equivalence and performance. Equivalence is assessed using an off-the-shelf tester or verifier, whereas performance is evaluated either by actual execution or by estimation through a database cost model. It also updates the selected rules and their benefits in the NLR2 repository. With each iteration, a larger set of optimized queries and a more comprehensive collection of NLR2s with more precise benefit estimates are obtained. Eventually, once the set of optimized queries stabilizes (or the budget is exhausted), the rewritten queries are presented to the user for final inspection.

Algorithm 1: GenRewrite top-level Algorithm

Input: $\mathcal{Q}$: queries, $\mathcal{L}$: LLM, $\mathcal{D}$: database, $\mathcal{T}$: tester, B:budget, θ: user-specified minimum desirable speedup
Output: $Res = \{\langle q, q', e\rangle | \forall q \in \mathcal{Q}_{opt} \subseteq \mathcal{Q}$, where $\mathcal{Q}_{opt}$ is the optimizable subset of $\mathcal{Q}\}$: Optimized queries & associated NLR2s

```
Res ← {};
NLR2 Repostitory R ← {} or pre-built repository R_pre;
while Q is not empty do
    for each query q in Q do
        q̃, e ← Suggest-and-explain (q, L, R);
        q' ← Correct-for-equivalence (q, q̃, L, D);
        equiv, speedup ← Evaluate-rewrite (q, q', D, T);
        if equiv is true then
            R ← Update-NLR2-repo (R, e, speedup);
            if speedup > θ then
                Res.add ( ⟨q, q', e⟩ );
    if Res does not change or budget B is exhausted then
        return Res;
    Remove queries in Res from Q;
```

4.1 Natural Language Rewrite Rules (NLR2s)

Since basic prompts do not consistently yield effective results, especially for highly complex queries, it is essential to incorporate proper hints into the prompt to guide the LLM. In GenRewrite, hints are represented as **Natural Language Rewrite Rules (NLR2s)**. The corresponding prompt format is presented in Figure 4.

	NLR2 as Rewrite/Transformation hints	Speedup
r1	Split aggregated computations by filtering on the specific period in separate subqueries	24.5x
r2	Combine multiple self-joins by pivoting aggregated data into columns using CASE	11.7x
r3	Eliminate union operations by computing aggregates in dedicated subqueries per data source	9x

Table 1. Natural Language Rewrite Rules (NLR2s) that benefit TPC-DS Q11 query with their corresponding speedups.

NLR2s encapsulate beneficial rewrites using natural language, differing fundamentally from traditional pattern-matching rewrite rules, which define a pattern-matching transformation and constraints specifying when that transformation is safe. Instead, NLR2s offer greater flexibility and expressiveness, as they are not restricted to matching fixed patterns but can capture nuanced, contextual insights into when and how a query should be rewritten.

Examples of NLR2s . Table 1 highlights three example NLR2s that yield significant speedups for Q11 (Listing 1). These transformations are mutually exclusive, each producing a distinct query structure and logical flow. Notably, the NLR2 addressing **`UNION ALL`** redundancy in Section 2 corresponds to r_3. The remaining two transformations leverage different early data filtering ideas, reducing the size of intermediate results and enhancing overall efficiency. None of these transformations can be captured by existing pattern-matching rules.

We meticulously curate a set of NLR2s in the NLR2 repository. These NLR2s fulfill a dual role: they act as both guiding hints for LLMs and explanations for users. When incorporated into the prompts for LLMs, these rules provide appropriate guidance for the rewriting direction. Meanwhile, presenting these NLR2s to users can help them to better understand the rationale behind the proposed modifications. With the prompt in Figure 4, LLMs prioritize implementing the provided NLR2s over relying on own creativity. The quality and relevance of these NLR2s directly impacts the effectiveness of the rewrites. However, manually analyzing each query and formulating effective NLR2s is impractical. Therefore, there is a need for an automated pipeline to identify promising NLR2s.

Collecting NLR2s. Our approach is simple yet effective: we prompt the LLM to generate candidate rewrites while simultaneously summarizing the underlying NLR2s. If a rewrite proves effective in subsequent evaluations, we consider its associated NLR2 valuable and store it—along with the corresponding query features and observed speedup—in the NLR2 Repository for future use. We will discuss NLR2 evaluation and query feature extraction later. To elicit rewrite rules, we append the instruction "Describe the rewrite rules you are using" to the basic prompt template in Figure 3. Moreover, if extracted NLR2s contain concrete column or table names, they may degrade the quality of future rewrites. To mitigate this, we explicitly instruct the LLM to exclude query-specific details by adding "do not include any specific query details in the rules, e.g., table names, column names" to the prompt, ensuring the NLR2s remain as general and reusable as possible.

Measuring the Benefit of NLR2s. In traditional query optimization, multiple rewrite rules can be applied on the same query as long as the changes they make do not conflict with each other. Similarly, we may collect multiple NLR2s associated with one candidate rewrite q'. In our experiments on the TPC-DS benchmark, we observe that each rewrite typically corresponds to 3–6 NLR2s. However, both generating rewrites with LLMs and executing complex queries can

r_a	Replace the correlated subquery with a precomputed aggregate in a separate CTE
r_b	Use common table expressions (CTEs) to precompute aggregates
r_c	Replace a correlated subquery that computes an aggregate with an inline view (or common table expression) that pre-aggregates data
r_d	Replace a correlated subquery with a CTE for aggregated values

Table 2. Examples of a set of NLR2s that are semantically equivalent to each other

be computationally expensive. It is therefore essential to prioritize NLR2s that yield meaningful performance gains and avoid wasting resources on ineffective ones.

We observe two key challenges that hinder efficient and accurate estimation of an NLR2's benefit.

Observation 1. A single rewrite q' is often associated with multiple NLR2s, each affecting query performance differently. This makes it challenging to pinpoint the precise contribution of each rule to any observed performance improvement. Assuming that all NLR2s associated with the same rewrite are equally effective is unrealistic and could undermine the effectiveness of our prompting strategy.

Observation 2. Two NLR2s can be functionally identical despite having different descriptions. For example, the NLR2s in Table 2 all target the same performance issue caused by correlated subqueries. Failing to recognize such equivalence can lead to redundant evaluations, wasting both resources and time.

We introduce two techniques to address the issues mentioned above: ***plan-based dominant NLR2 identification*** and ***NLR2 grouping via LLMs***.

Plan-based dominant NLR2 identification. When the LLM produces a faster rewrite q' for a given input query q, it also returns a set of associated NLR2s, denoted as $\{r_1, r_2, \ldots, r_k\}$. Instead of treating all rules equally, we aim to identify a single dominant NLR2, denoted r^*, that primarily accounts for the observed performance gain. This allows us to attribute the speedup to the most impactful transformation, rather than distributing credit uniformly across all contributing rules. To identify the dominant NLR2 r^*, we prompt the LLM with each NLR2 individually on the same input query q, producing candidate rewrites $q_{r_1}, q_{r_2}, \ldots, q_{r_k}$. For each q_r, we obtain its query plan $P(q_r)$ via `EXPLAIN` (which reports the plan without executing the query) and compare it to the plan of q', denoted $P(q')$. We first check for an exact cost match: if there exists a rule r such that the optimizer's cost estimate for q_r equals that of q' (i.e., $C(q_r) = C(q')$), we set $r^* = r$, since identical costs typically indicate identical plans. Otherwise, we choose the NLR2 whose plan is most similar to $P(q')$:

$$r^* = \arg\min_{r \in \mathcal{R}} d_{\text{plan}}\left(P\left(q_r\right), P\left(q'\right)\right)$$

where plan similarity is measured by embedding each `EXPLAIN` plan using `bert-base-uncased` and taking the ℓ_2 distance between the embeddings: $d_{\text{plan}}\ (P_1, P_2) = \|\phi\left(P_1\right) - \phi\left(P_2\right)\|_2$. We then attribute the observed speedup of q' over q to r^* and treat that improvement as the estimated benefit of the dominant rule.

The identification process relies on exactly one piece of execution, i.e., the improved rewrite q', which serves as the anchor. We do not execute any additional candidate rewrites $q_{r_1}, q_{r_2}, \ldots, q_{r_k}$; instead, we rely on inexpensive plan and cost comparisons. This assumption of a single dominant NLR2 rewrite simplifies the real-world scenario, where interactions between rules can occur. In practice, a single rule may yield little benefit in isolation but unlock further optimizations when

Input:
Replace implicit joins with explicit joins ←the incoming NLR2
Please select the rewrite rule that is strictly the same as the above rule and give your explanation (just give one answer). If not, please select the first item "Unseen rule".
Options:
1. Unseen rule
2. Replace subqueries with join
3. Use explicit join syntax instead of comma-separated tables in the from clause
4. Use a CTE to calculate the average
Answer:
Use explicit join syntax instead of comma-separated tables in the from clause ←LLM's choice after reasoning
Explanation: Both the original rule and this rewritten rule are about replacing implicit joins with explicit joins. **Implicit joins are indicated by comma-separated tables in the FROM clause and join conditions in the WHERE clause**. ...

Figure 5. The prompt to predict the group to which the incoming NLR2 belongs

combined with others. However, exhaustively evaluating all rule combinations is computationally prohibitive. Our method offers a scalable approximation.

NLR2 grouping via LLMs. While plan-based dominant NLR2 identification efficiently leverages query execution history to avoid running expensive queries on real data, repeatedly invoking the LLM on q with different NLR2s as hints incurs a noticeable amount of API cost. As noted in **Observation 2**, the set of NLR2s extracted from a single query q are often not sufficiently distinctive. Supplying semantically equivalent hints to the LLM yields little additional benefit while unnecessarily increasing the costs. To address this, every time a new NLR2 r_{new} is discovered for query q, we leverage LLMs to determine its semantic equivalence to existing NLR2s associated with q. If it is deemed equivalent to a previously identified group, we avoid invoking the LLM again for that rule; otherwise, a new group is created for r_{new}, and we proceed to run the LLM rewriting pipeline to generate the corresponding rewrite $q_{r_{\text{new}}}$ and obtain its query plan. This allows us to evaluate whether r_{new} qualifies as the dominant NLR2 for query q.

Figure 5 illustrates an example prompt used for this grouping process. Replacing implicit joins with explicit joins hardly yields any noticeable performance gap, yet such transformations are often included among the collected NLR2s. Grouping NLR2s not only prevents redundancy in our repository but also simplifies maintenance and ensures that our efforts are concentrated on rewrite strategies that deliver substantial performance gains.

4.2 Suggesting Candidate Rewrites

In the previous section, we discussed how to construct a high-quality NLR2 Repository over time. The primary goal of this repository is to enable effective knowledge transfer, particularly for optimizing complex queries that vanilla LLM-based approaches fail to handle. However, NLR2s that yield substantial speedups for certain queries may degrade performance when applied to others. A key challenge, therefore, is to identify queries that share similar performance bottlenecks, so that we can selectively incorporate the most relevant NLR2s from the repository into the prompt for effective and equivalent rewriting.

Query performance bottleneck analysis. Since we focus on optimizing recurring queries with readily available query plans, we aim to extract as much insight as possible from these plans. To this end, we leverage LLMs to inspect query plans and identify potential performance bottlenecks. Prior to analysis, we preprocess each plan to remove irrelevant fields, thereby reducing the number

of tokens and minimizing distractions from non-essential details. For example, in PostgreSQL, eliminating over 20 unnecessary fields can reduce token count by up to 43%. We then provide the LLM with the prompt consisting of three components: the original query q, the trimmed query plan, and a concise instruction: "Summarize the most critical performance bottleneck based on the query and its plan." It is important to note that the query plans used for bottleneck analysis are retrieved from execution history and include actual runtime metrics (e.g., actual time, rows, loops) for each node. In contrast, the plans used for identifying dominant NLR2s only include the planner's chosen execution plan and its estimated costs, without any observed runtime statistics.

Similarity search. Using the bottleneck summary generated by the LLM, we perform a two-stage similarity search to identify relevant past queries. (1) we encode the bottleneck summary into a feature embedding and retrieve the top-k most similar embeddings from previously analyzed queries. (2) we apply a refinement step inspired by the method illustrated in Figure 5: we prompt the LLM with a multiple-choice question to identify which of the retrieved candidates shares the most similar performance bottleneck with q. If the LLM determines that none of the candidates are a good match, the system falls back to using the basic prompt without incorporating any prior NLR2. Based on the selected candidate, we then retrieve and apply its most beneficial associated NLR2.

Input:
q1: [Input query]
q2: [Candidate rewrite]
q1 is the original query, q2 is the rewritten query of q1.
For q1, break it down step by step and then describe what it does in one sentence. Do the same for q2.
Give an example, using tables, to show that these two queries are not equivalent if there's any such case. Otherwise, just say they are equivalent.
Answer:
Not equivalent.
[Breakdown and analysis with **counterexamples**]
Input:
Based on your analysis, which part of q2 should be modified so that it becomes equivalent to q1? Show the modified version of q2.
Answer:
[Revised candidate rewrite]

Figure 6. The prompt for semantic correction

4.3 Correcting Candidate Rewrites

Unlike traditional rewrite rules, which are verified for correctness, LLM-generated rewrites lack guaranteed accuracy, even with appropriate rewrite hints. While many tools can assess query equivalence with varying degrees of capability and efficiency, our requirements extend beyond this. Our goal is not only to verify the equivalence of the rewrite to the input query but also to correct the rewrite until it matches the original query. Otherwise, it is inefficient and a waste of computing resources to repeatedly prompt the LLM in hopes of producing a correct rewrite.

Errors generally fall into two categories: 1) Executable with incorrect outcomes, such as a mismatch where the input query aims to find the second-highest salary, whereas the candidate rewrite seeks the highest salary—this type of error is termed a semantic error. and 2) Unexecutable due to various issues, such as incorrect column names or table names—this type is referred to as a

syntax error. These two categories of errors are relatively independent of each other. Inspired by these observations, we develop a two-stage counterexample-guided correction approach. During the semantic correction stage, we focus on ensuring that the candidate rewrite achieves the same intent as the input query, while temporarily setting aside syntax errors. In the syntax correction stage, we address all issues impacting query execution.

Semantic Correction. Given the input query as the gold standard, we iteratively refine each candidate rewrite to ensure convergence toward semantic equivalence. In each iteration, the LLM first summarizes the intent and logic of both the original and candidate queries to understand their respective semantics. It then attempts to identify a counterexample that highlights a divergence in their results, leveraging the extracted summaries and reasoning. Based on this comparison, the LLM determines whether the candidate rewrite is equivalent to the input query. If deemed "not equivalent", the LLM generates a revised version of the candidate query, informed by the identified discrepancies. This revised candidate is evaluated in the next iteration. The process continues until the LLM deems the candidate query equivalent to the input. Figure 6 illustrates an example prompt used in the semantic correction phase.

Syntax Correction. Syntax correction is also an iterative process, similar to semantic correction, but with a key distinction: syntax correction depends on feedback from executing the `EXPLAIN` command on the candidate rewrite to guide its refinement process, while semantic correction relies on counterexamples identified by the LLM through logical reasoning. At each iteration, we prompt the LLM with the original query q, the candidate rewrite q', and the error message returned by the `EXPLAIN` command (if any). Typical errors addressed include incorrect column or table names and inconsistent table aliases. The use of EXPLAIN, which does not execute the query, minimizes overhead, making it an efficient method for deciding whether the rewrite is executable or not.

4.4 Evaluating Candidate Rewrites

Our ultimate goal is to identify equivalent queries that are *faster*, without introducing regressions. To this end, we subject each candidate rewrite to two automated gates: a correctness gate and a performance gate.

Correctness gate. We employ a combination of off-the-shelf verifiers and testers, selected based on the available time budget and the query complexity. Specifically, we use SQLSolver [14], a state-of-the-art SQL verifier, to assess the correctness of generated rewrites. If SQLSolver returns "UNKNOWN", we fall back to a tester from SlabCity [15], which extracts execution hints from the input query and applies syntactic analysis to guide input generation. Counterexamples generated during the semantic correction phase can also be incorporated into the tester to enhance coverage. If neither the verifier nor the tester can certify equivalence, we abstain and discard the rewrite.

Performance gate. Only candidates that pass the correctness gate proceed to performance evaluation. We run an offline shadow execution of the candidate rewrite against the target database. A rewrite is accepted only if it satisfies a "never-worse-off" policy; otherwise, the rewrite is automatically rejected and the system falls back to the original query. We also pre-filter with a static cost proxy to avoid expensive trials. This procedure is particularly justified for workloads with recurring queries of fixed periodicity.

In the exploratory setting, an optional manual review step can be used to expand coverage, but this is not required for the deployment.

5 Evaluation

The evaluation aims to answer the following questions:

(1) How does GenRewrite compare to a straighforward application of the out-of-the-box LLM? In other words, do our NLR2s and counterexample-based correction algorithm considerably boost the LLM's performance? (§5.2)
(2) How does GenRewrite compare to state-of-the-art rule-based query rewriting approaches? Specifically, which approach can optimize more queries and which approach yields more speedup for the rewritten queries? (§5.3)
(3) How much does each of GenRewrite's design contribute to its overall success (ablation study)? (§5.4)
(4) What is the time and monetary cost of GenRewrite? (§5.5)

5.1 Experimental Setup

Testbed. Our experiments are all conducted on Google Cloud Engine Machine n2-highmem-2 instances, with with 16GB RAM and 2 vCPUs. We use PostgreSQL 14.17. To evaluate query latency accurately, we execute each query 3 times and take their average as the final latency value. We do not create any indexes. Instead, we run `VACUUM ANALYZE` on all tables before benchmarking, which collects statistics for cost-based plan selection.

Choice of LLM. We designate OPENAI o3-mini as the default LLM, due to its cost efficiency and robust reasoning capabilities. We also assess GenRewrite's generalizability across other LLMs with varying capabilities, such as GPT-4o and GPT-4-mini.

Workloads. Our experiments include two widely used analytical benchmarks, TPC-DS [7] and the Join Order Benchmark (JOB) [23], as well as additional query sets generated by SQLStorm [36] over the same TPC-DS and JOB schemas. SQLStorm uses LLMs to synthesize thousands of executable, high-complexity SQL queries and then filters them using actual database feedback. These SQLStorm queries were released in June 2025—after the pretraining cutoff of the LLM we use (October 2023)—and thus were not available during pretraining, allowing us to assess how well GenRewrite generalizes to previously unseen queries on familiar schemas.

(1) **TPC-DS.** TPC-DS is an industry-standard decision support benchmark. While it does not perfectly capture the recurring, user-specific workloads seen in production, it remains a strong proxy for realistic analytical queries and is widely used to compare query optimization techniques. We generated a 10 GB TPC-DS instance and used all 99 benchmark queries.
(2) **Join Order Benchmark (JOB).** JOB is a canonical benchmark derived from the IMDB dataset and is heavily used to study join ordering and cost-based optimization. Its queries span diverse levels of complexity and involve multi-way joins, making it a good stress test for optimizer behavior.
(3) **TPC-DS (SQLStorm).** SQLStorm generated tens of thousands of query variants over the TPC-DS schema. Running LLM-based methods on all of them would be prohibitively expensive due to LLM inference cost. We therefore select the 100 slowest SQLStorm-generated queries as measured by runtime under `olapbench` [4]. According to SQLStorm's own analysis, these queries fall in the medium-to-high complexity range.
(4) **JOB (SQLStorm).** We apply the same selection criterion for JOB: We select the 50 slowest queries by runtime under `olapbench`. As with the TPC-DS (SQLStorm) set, these queries are classified as medium to high complexity.

Baselines. We compare GenRewrite against two LLM-based approaches and three state-of-the-art rule-based query rewriting methods, two of which, LLM-R^2 and R-Bot, incorporate LLMs:

- **Baseline LLM** employs a basic prompting technique illustrated in Figure 3 to generate candidate queries without any correction mechanism. This baseline represents the straightforward application of out-of-the-box LLM.
- **LITHE** [13] is an LLM-based method that constructs a database-sensitive prompt by selecting the most appropriate rule for the input query from a fixed set of six handcrafted rules. These rules target redundancy elimination and selectivity improvement.
- **LLM-R^2** (VLDB 2025 [27]) is an LLM-enhanced query rewrite system that can automatically select effective rules from a given set of rewrite rules to rewrite an input SQL query by leveraging a high-quality demonstration pool.
- **R-Bot** (VLDB 2025 [41]) is a query rewrite system that retrieves the most relevant rewrite evidences and employs an LLM-driven, step-by-step method to iteratively select and arrange rewrite rules based on retrieved evidence.
- **LearnedRewrite** (or **LR**, VLDB 2022 [47]) is a state-of-the-art rule-based query rewriter [47] which uses Monte Carlo Tree Search to guide the rule-based rewriting process. LR uses rewrite rules from Calcite [10], which is the most popular library for query rewriting rules.

Note that while both LLM-R^2 and R-Bot utilize LLMs to enhance rule selection and sequencing, they do not explore beyond the space defined by the existing rule set. Therefore, we still classify them as rule-based approaches. We excluded [44] from our experiments as it was not able to produce valid rewrites for any TPC-DS queries.

Setup. Rule-based query rewriting methods produce the same rewrite for an input query across multiple runs. In contrast, due to the statistical nature of LLMs, additional LLM runs increase the likelihood of uncovering new rewrite possibilities and thereby improving performance. To ensure a fair comparison across approaches, we fix the number of iterations for each LLM-based method to four, generating four rewrite candidates per query. Baseline LLM uses the basic prompt (see Figure 3) in all four iterations, without any correction or contextual enhancement. GenRewrite also uses the basic prompt in the first iteration, as the NLR2 Repository is initially empty and we do not use pre-seeded or external rules. In subsequent iterations, it selectively incorporates relevant and beneficial NLR2s retrieved from the repository, if such rules are available, to guide the rewriting process more effectively. The semantic- and syntax-correction loops are each capped at 3 iterations. If, after these caps, a candidate still fails semantic correction or remains unparsable by the DBMS, we discard it and retain the original input q. A method is considered successful on the given query q if any of these four candidates is both equivalent to the original query and faster. If any method, rule-based or LLM-based, fails to produce an equivalent rewrite for the given query, we assign a speedup ratio of 1, indicating that the original query is used as a fallback.

5.2 Comparison with LLM-based Methods

Table 3 compares the fraction of queries that each method speeds up beyond several thresholds. Before discussing individual results, we note an important trend: queries in JOB are generally less complex than those in TPC-DS, both in the original benchmarks and in the SQLStorm-generated variants. In the original workload, JOB queries average 278.81 tokens, whereas TPC-DS queries average 441.34 tokens (a 58% increase). Although JOB queries can involve many joins and are therefore challenging for join ordering, their overall structure is simpler: they contain fewer nested subqueries, window functions, and set operations than TPC-DS. This gap persists in the SQLStorm workloads. Even when SQLStorm generates new queries with medium to high complexities over both schemas, the JOB-derived queries remain constrained by narrower, more normalized tables and by the absence of complex cross-channel or temporal comparisons. This difference in query complexity translates to different optimization potential. In TPC-DS and its SQLStorm variants,

speedup	TPC-DS >2x	TPC-DS >10x	TPC-DS >50x	JOB >1.2x	JOB >2x	JOB >10x
GenRewrite	25 (25.3%)	9 (9.1%)	6 (6.1%)	20 (17.7%)	8 (7.1%)	3 (2.7%)
Baseline LLM	18 (18.2%)	6 (6.1%)	5 (5.1%)	19 (16.9%)	7 (6.2%)	3 (2.7%)
LITHE	16 (16.2%)	6 (6.1%)	4 (4.0%)	17 (15.0%)	6 (5.3%)	1 (0.9%)
LLM-R^2	10 (10.1%)	6 (6.1%)	5 (5.1%)	9 (8.0%)	2 (1.8%)	1 (0.9%)
R-Bot	9 (9.1%)	5 (5.1%)	3 (3.0%)	7 (6.2%)	3 (2.7%)	2 (1.8%)
LR	6 (6.1%)	4 (4.0%)	0 (0%)	6 (5.3%)	2 (1.8%)	1 (0.9%)

(a) Original workloads.

speedup	TPC-DS (SQLStorm) >2x	TPC-DS (SQLStorm) >10x	TPC-DS (SQLStorm) >50x	JOB (SQLStorm) >1.2x	JOB (SQLStorm) >2x	JOB (SQLStorm) >10x
GenRewrite	25 (25.0%)	13 (13.0%)	3 (3.0%)	7 (14.0%)	6 (12.0%)	1 (2.0%)
Baseline LLM	12 (12.0%)	9 (9.0%)	3 (3.0%)	4 (8.0%)	2 (4.0%)	1 (2.0%)
LITHE	19 (19%)	11 (11.0%)	2 (2.0%)	6 (12.0%)	4 (8.0%)	1 (2.0%)
LLM-R^2	2 (2.0%)	1 (1.0%)	0 (0.0%)	2 (4.0%)	2 (4.0%)	0 (0.0%)
R-Bot	2 (2.0%)	0 (0.0%)	0 (0.0%)	2 (4.0%)	2 (4.0%)	0 (0.0%)
LR	4 (4.0%)	1 (1.0%)	0 (0%)	3 (6.0%)	1 (2.0%)	0 (0.0%)

(b) SQLStorm-generated workloads.

Table 3. Speedup comparison beyond thresholds between all methods. Upper: original TPC-DS and JOB. Lower: SQLStorm-generated TPC-DS and JOB. We report absolute counts and percentages to reflect the different workload sizes.

GenRewrite can optimize roughly 25% of queries by at least 2x, whereas in JOB and its SQLStorm variant, only about 7%–12% of queries reach that level of improvement.

Overall, GenRewrite delivers the strongest improvements across all benchmarks, and this advantage is most pronounced on the more complex TPC-DS workloads. LLM-based methods outperform rule-based query rewriting approaches by notable margins. On the original TPC-DS benchmark, GenRewrite speeds up 25 queries by at least 2x and 9 queries by at least 10x. By comparison, Baseline LLM speeds up 18 and 6 queries at those thresholds, and LITHE speeds up 16 and 6 queries. On the SQLStorm-generated TPC-DS workload, GenRewrite again leads: it speeds up 25 queries by at least 2x and 13 queries by at least 10x, compared to 12 and 9 for Baseline LLM and 19 and 11 for LITHE. On JOB and JOB (SQLStorm), all three methods show more limited gains, since these queries offer less optimization opportunity. GenRewrite still exhibits a slight edge, and the gap between methods is less pronounced than on TPC-DS.

An interesting finding is that Baseline LLM outperforms LITHE on the standard TPC-DS benchmark, but the ranking flips on TPC-DS (SQLStorm); we observe the same trend on JOB vs. JOB (SQLStorm). This suggests that Baseline LLM benefits from the LLM's prior exposure to

well-known public benchmarks and can often reproduce established optimizations there. On newly generated, previously unseen queries (e.g., TPC-DS (SQLStorm)), however, Baseline LLM both struggles to suggest a good rewriting direction and more frequently produces invalid SQL. LITHE is more robust in that setting because it explicitly chooses one of six handcrafted optimization rules (e.g., redundancy removal or increased selectivity) and conditions the LLM on that rule plus an example rewrite; this often gives the model a strong starting direction. But LITHE has two structural limitations: it cannot repair incorrect rewrites, and its handcrafted rule set cannot evolve as new workloads arrive. GenRewrite addresses both issues: it uses counterexample-guided iterative correction to fix invalid or semantically inequivalent rewrites, and it transfers knowledge via NLR2s that accumulate and improve over time, increasing the likelihood that the generated rewrite is both correct and faster.

Table 4 compares performance of GenRewrite and Baseline LLM in terms of the correctness of generated rewrites. Both methods are nearly perfect on JOB, where query complexity is low. On TPC-DS benchmark, GenRewrite produces at least one equivalent rewrite for 85.9% of queries, outperforming Baseline LLM 's 74.7%. Moreover, 70% of GenRewrite 's generated rewrites are correct, compared to only 51.8% for Baseline LLM. Notably, approximately one-third of Baseline LLM 's rewrites fail to execute due to minor syntax errors or fabricated table/column names, while only 13.3% of GenRewrite 's rewrites suffer from such issues, highlighting the effectiveness of our iterative correction module. These correctness differences matter for performance. Because Baseline LLM only generates a correct rewrite about half the time, its ability to suggest a promising optimization direction does not always translate into a usable speedup. Table 3 shows that GenRewrite reliably finds more faster rewrites than Baseline LLM within the same four rewrite iterations. This does not imply that Baseline LLM is inherently incapable of finding faster rewrites for them; rather, it may require substantially more iterations and LLM API calls to do so. However, the number of additional runs needed is unpredictable, making the approach less practical in real-world settings. In contrast, GenRewrite uses NLR2s to carry forward rewrite knowledge from previously optimized queries and incorporates performance feedback into subsequent prompts, increasing the likelihood of discovering effective rewrites in fewer iterations.

While GenRewrite and Baseline LLM achieve similar speedup for more than half of these representative queries, there are a few interesting cases where GenRewrite significantly outperforms Baseline LLM. These cases highlight the value of incorporating NLR2s in GenRewrite.

Case 1: Both can optimize Q4, Q11, Q74, but GenRewrite achieves much better speedup. GenRewrite infers from the query plan that Q4, Q11, and Q74 likely share similar bottlenecks, specifically, issues such as "*Excessive nested loop joins on CTE scans*" and "*Disk-based sort and aggregate operations on large fact tables*". Consequently, these queries can benefit from analogous optimization strategies. More specifically, the NLR2 "*Replace multiple UNION subqueries with separate aggregation subqueries using CASE expressions*" is derived from Q74, while the NLR2 "*Split aggregated computations by filtering on the specific period in separate subqueries*" originates from Q11. Both of these transformations yield a higher speedup for Q4 compared to its original effective NLR2: "*Replace multiple self-joins with pivoted aggregations*" (18x, 7x vs. 3.5x). Notably, the NLR2 from Q11 produces more consistent improvements for Q4, as the transformation is relatively straightforward and encounters fewer issues of inequivalence.

Case 2: Baseline LLM cannot find equivalent rewrites for Q23 while GenRewrite can. LLMs are trained on vast datasets and accumulate substantial domain knowledge, which often enables them to perform intelligent optimizations. However, in some cases an LLM's chosen strategy can be flawed, consistently yielding inequivalent outputs. For example, Listing 2 shows two queries (irrelevant parts omitted for clarity): on the left, TPC-DS Q23, and on the right, a

		GenRewrite	Baseline LLM
TPC-DS	% of input queries with ≥ 1 equiv. rewrite	85.9%	74.7%
	% of rewrites that are equivalent	70.0%	51.8%
	% of rewrites that are inequivalent	13.4%	13.1%
	% of rewrites that are undetermined	3.3%	2.0%
	% of rewrites that cannot be executed	13.6%	33.1%
JOB	% of input queries with ≥ 1 equiv. rewrites	100%	100%
	% of rewrites that are equivalent	99.8%	98%
	% of rewrites that are inequivalent	0%	0.7%
	% of rewrites that are undetermined	0%	0%
	% of rewrites that cannot be executed	0.2%	1.5%

Table 4. Equivalence checking results on original TPC-DS and JOB benchmark for GenRewrite and Baseline LLM. For each input query, both methods generate four candidate rewrites. The first row shows the percentage of input queries for which at least one equivalent rewrite was found. The remaining rows classify all generated rewrites (across all four attempts) by outcome: equivalent, inequivalent, undetermined, or failed to execute.

candidate rewrite generated by Baseline LLM, which consistently produces a similar query pattern over multiple rounds. Although the rewrite appears plausible, a subtle difference leads to divergent result sets. Specifically, the rewritten query joins `web_sales` with a common table expression (CTE) such that qualifying items may match more than once, duplicating rows and inflating the final sum. In contrast, the corresponding GenRewrite rewrite employs an IN predicate on `item_sk` from `frequent_items`. Even if the CTE contains duplicate rows, the **`IN`** clause merely checks for membership and does not multiply matches. This difference in join behavior explains why the two queries yield markedly different totals when executed at scale. GenRewrite adopts the NLR2 "*Pre-aggregate the CTE so it is computed once and returns fewer rows*" from Q95, achieving 3.34x speedup. This case highlights the importance of exercising explicit control over rewriting directions through the application of NLR2s, rather than relying on LLMs as black boxes. It also emphasizes the role of rule separation in facilitating effective and accurate knowledge transfer across queries.

```
...
select sum(sales)
...
from ...
web_sales
where ws_item_sk in
   (select item_sk
        FROM frequent_items)
...
```

```
...
select sum(sales)
...
from ...
web_sales as ws
   join frequent_items as f
   on ws.ws_item_sk = f.item_sk
...
```

Listing 2. Left: Original TPC-DS Q23 query. Right: A rewrite of Q23 generated by Baseline LLM that is inequivalent.

5.3 Comparison with Rule-based Methods

Analysis of Table. 3 shows that GenRewrite consistently surpasses the three rule-based baselines in terms of the number of queries optimized, regardless of the desired speedup threshold, on all benchmarks. Other LLM-based methods also consistently outperform rule-based methods with smaller margins than GenRewrite. We first consider the standard public benchmarks (Table 3a). On the original TPC-DS workload, GenRewrite identifies substantially more optimization opportunities than the rule-based baselines: it achieves at least 2x speedup on 25 queries, compared

to 10, 9, and 6 queries for LLM-R^2, R-Bot, and LR, respectively. The advantage persists at higher thresholds (e.g., 10x speedup), and we observe a similar pattern on JOB. These results suggest that GenRewrite already outperforms established rule-based methods on well-studied public benchmarks.

SQLStorm workloads (Table 3b) yield two key observations:

Observation 1. The performance gap between GenRewrite and the rule-based baselines widens on TPC-DS (SQLStorm). All three rule-based baselines (LLM-R^2, R-Bot, and LR) are built on top of Calcite and ultimately rely on Calcite's rewrite rules. Their differences lie in rule selection and ordering, but they inherit the same two Calcite limitations: (1) *Parser coverage:* The Calcite parser and validator do not fully support several constructs that appear in SQLStorm-generated TPC-DS queries, such as recursive CTEs, `LENGTH`, etc. As a result, these systems only emit rewrites for roughly half of the input queries. (2) *Rule coverage:* Even when Calcite can parse the query, its fixed rule set often fails to match the structure of these queries. SQLStorm's queries are not hand-polished benchmark SQL; they contain noisy nesting, ad hoc join predicates, and unseen operator combination that does not line up cleanly with Calcite's stock transformations (e.g., predicate pushdown). In these cases, the systems may not show meaningful improvement. In contrast, GenRewrite does not depend on Calcite's rule inventory. It can generate and iteratively refine rewrites for these same SQLStorm queries, which explains the widening gap on TPC-DS (SQLStorm).

Observation 2. On SQLStorm workloads, the "LLM-enhanced" rule-based systems degrade to the point that they can be outperformed by LR, which does not use an LLM at all. LLM-R^2 selects a few high-quality demonstration rewrites from a curated library using a learned retriever, and R-Bot retrieves offline "rewrite recipes" mined from past queries. These strategies work on TPC-DS and JOB because the benchmark queries follow canonical patterns that are well represented in those demonstration pools. Under SQLStorm, however, the generated queries deviate substantially from those canonical patterns, so the retrieved demonstrations and recipes are often no longer relevant and provide poor guidance. GenRewrite, by contrast, is not limited to a static set of expert demonstrations or a fixed rule inventory. It employs iterative correction and accumulates transferable rewrite knowledge from prior queries, enabling it to adapt to new query forms without manual rule engineering.

GenRewrite outperforms existing baselines both on the original TPC-DS / JOB benchmarks and on the SQLStorm-generated workloads. Its advantage becomes larger precisely in the regimes where fixed-rule systems provide poor coverage. If new rules were distilled from GenRewrite's successful rewrites on complex queries and then incorporated into Calcite, the rule-based methods could be strengthened and extended to a broader class of queries.

A deeper look at rules discovered by GenRewrite. Besides standard Calcite rules (e.g., predicate pushdown, projection pruning), several papers propose more aggressive rewrite patterns for analytical workloads. Generalized Sub-Query Fusion [35] eliminates redundant I/O by fusing multiple similar subqueries into a single shared scan/aggregation. Super-operators [22] collapse multi-branch aggregates and unions into a single streaming physical operator. The Spark engine in Azure Synapse [29] rewrites plans to reuse shuffles, push down filters and partial aggregates, and inject runtime partition pruning to reduce data movement. Athena [11] detects repeated expensive subplans in a query and fuses them so they are computed once and shared by all consumers. In Table 5, for each major rewrite pattern discovered by GenRewrite we ask: (1) does it align with any of these known patterns, at the idea level; and (2) is there an exact structural match already reported in prior work?

Representative optimization patterns discovered by GenRewrite:

Pattern	Covered queries	Similar in idea	Exact match
1	4, 11, 74	[22], [11]	No
2	10, 35, 69	[10]	No
3	41	[10]	No
4	23998*		No
5	22722*	[35], [11]	No
6	61, 88	[35], [11]	Yes, [11]
7	28982*, 34389*	[10]	Yes, [10]

Table 5. Summary of rewrite patterns discovered by GenRewrite. "Covered queries" lists the queries that exhibit each pattern (queries with * are from TPC-DS SQLStorm; others are from standard TPC-DS). "Similar in idea" notes prior work that proposes a conceptually related optimization. "Exact match"indicates that the same structural rewrite has been reported before.

	TPC-DS				
	>2x	>10x	>50x	avg. time(s)	avg. cost($)
GenRewrite	25 (25.3%)	9 (9.1%)	6 (6.1%)	151.82	0.129
GenRewrite without grouping NLR2s	25 (25.3%)	9 (9.1%)	6 (6.1%)	385.47	0.310
GenRewrite without bottleneck analysis	19 (19.2%)	6 (6.1%)	5 (5.1%)	150.30	0.124
GenRewrite with gpt-4o	14 (14.1%)	5 (5.1%)	5 (5.1%)	105.40	0.117
GenRewrite with gpt-4-mini	2 (2.0%)	1 (1.0%)	0 (0.0%)	301.20	0.016

Table 6. Ablation study: impact of GenRewrite's design decisions on LLM runtime, cost, and the share of queries achieving various speedups.

(1) Pivot with conditional aggregation instead of UNION-then-self-join.
(2) Replace multiple `EXISTS` conditions with a precomputed set via joins.
(3) Rewrite `COUNT(*)>0` as `EXISTS` (aggregate → semi-join; enables short-circuiting).
(4) Replace a bounded self-recursive CTE with a non-recursive base query `CROSS JOIN`ed to a small numbers table.
(5) Lift a per-key summary (e.g., item-level totals) into one CTE and drive all branches from it.
(6) Consolidate repetitive subqueries into a single aggregation subquery with conditional expressions.
(7) Decorrelate `IN` by joining a `SELECT DISTINCT` key,... mapping table instead of re-checking the subquery.

5.4 Detailed Analysis and Ablation Study

In this section, we conduct a detailed ablation study to validate our design decisions in GenRewrite and the contribution of each component therein towards the overall performance. We focus primarily on the original TPC-DS benchmark, as its complexity provides a challenging test of our system's capabilities. The results, summarized in Table 6, highlight the impact of individual components on LLM runtime overhead, monetary cost, and the percentage of queries achieving various levels of speedup.

Plan-based dominant NLR2 identification. This component plays a critical role in reducing query execution overhead, although its impact is not directly reflected in Table 6. Without plan-based dominant NLR2 identification, we would need to execute each rewrite—generated by applying

	k=1		k=2		k=3		k=4	
	eq	fast	eq	fast	eq	fast	eq	fast
No correction	52	5	67	8	70	12	74	21
Semantic only	54	5	68	10	71	14	75	21
Semantic + syntax	**70**	**5**	**79**	**10**	**85**	**16**	**86**	**28**

Table 7. Cumulative count of queries with (i) at least one equivalent rewrite (*equiv*) and (ii) at least one equivalent rewrite that is at least 20% faster than the original (*faster*), as the candidate budget increases (k=1 . . . 4), under different correction strategies.

a single NLR2—to measure its runtime, which can be prohibitively expensive for long-running queries.

NLR2 grouping via LLM. We collect an average of 5.08 NLR2s per input query after applying grouping, compared to 13.24 NLR2s per query without grouping. Although we only re-evaluate the impact of an NLR2 when the associated performance improvement exceeds a predefined threshold, grouping still significantly reduces both monetary and time costs. Without grouping, the average LLM runtime increases by 153.9%, and the monetary cost rises by 140.3%, demonstrating that NLR2 grouping's impact on costs.

Query performance bottleneck analysis. We compare two strategies: (1) apply the top NLR2s globally, ignoring query similarity; and (2) our default, which retrieves candidates by bottleneck similarity. The global strategy performs essentially the same as Baseline LLM, indicating that "best overall" strategy rarely transfers. Bottleneck-aware retrieval, in contrast, delivers clear gains: effective reuse hinges on aligning NLR2s with performance bottlenecks.

Choice of the LLM. GenRewrite's performance critically depends on the LLM's reasoning capabilities. We evaluate three models: OpenAI's `o3-mini` (reasoning-oriented), `gpt-4o` (balanced), and `gpt-4o-mini`(most cost-efficient). GenRewrite powered by `gpt-4o` is noticeably outperformed by the `o3-mini` variant. We decompose GenRewrite 's workflow into two stages: (1) discovering effective rewrites and summarizing NLR2s, and (2) applying high-quality NLR2s to guide rewrites. We observe that `gpt-4o` struggles primarily in stage 1. However, when GenRewrite with `gpt-4o` is supplied with NLR2s previously discovered by `o3-mini`, the number of queries achieving a speedup $\geq 10\times$ increases from 5 to 8, further demonstrating the effectiveness of NLR2-guided rewriting.

Parameter sensitivity analysis.

Budget: maximum candidates per query $K = 4$; correction caps $I_{\text{sem}} = I_{\text{syn}} = 3$ (semantic and syntax loops). For the TPC-DS workload, the average iterations is 0.30 (semantic) and 0.76 (syntax). The per-query correction iteration distributions are:

- Semantic: 0: 76.72%, 1: 19.02%, 2: 1.64%, 3: 2.62%.
- Syntax: 0: 60.66%, 1: 18.36%, 2: 5.25%, 3: 15.74%.

We vary the candidate budget $k \in \{1, 2, 3, 4\}$. Table 7 reports the *cumulative* number of queries that achieve (i) ≥ 1 equivalent rewrite and (ii) ≥ 1 equivalent rewrite that is at least 20% faster, as k increases (higher k means more candidates attempted per input query), comparing: (i) no correction, (ii) semantic-only correction, and (iii) semantic + syntax correction. Table 7 shows that correction loops markedly reduce the effective error rate, with *semantic + syntax* outperforming the other settings at every k. Larger k improves coverage with diminishing returns and proportionally higher cost.

Desired speedup threshold: We set θ to 1.2x in our implementation. Increasing θ reduces coverage but raises the average speedup among accepted rewrites. With a lower θ, we keep more rewrites — including many that are only modestly better.

Desired speedup threshold θ	1.2x	2.0x	5.0x
Coverage (%)	28.3	23.2	13.1
Geom. mean speedup	11.1x	17.0x	72.2x

Table 8. Coverage and geometric mean speedup for desirable speedup thresholds $\theta \in \{1.2x, 1.5x, 2.0x\}$. Coverage(%) is the fraction of queries with $\geq \theta$ speedup; geom. mean is over covered queries.

Counterexample-guided iterative correction. We compare our counterexample-based semantic check to Miniature & Mull from LLM-SQL-Solver [46] on random TPC-DS rewrites (no correction). Our prompt yields 29% equivalent rewrites before syntax correction with a 6% false-positive rate; Miniature & Mull achieves 26% equivalents with 22% false positives. Thus our prompt better aligns with the iterative correction task—more verified equivalents and far fewer false positives.

5.5 Time and Monetary Cost

While GenRewrite 's time and API costs are a one-time, amortized expense for recurring workloads (see Target workloads in §1), we still quantify them to ensure they remain modest. Using OpenAI's `o3-mini` to generate a rewrite for a TPC-DS query—including suggestion, semantic correction, and syntactic correction—costs on average $0.029 and takes 36.15 s. For comparison, `gpt-4o` and `gpt-4o-mini` average ($0.026, 25.1 s) and ($0.004, 71.71 s), respectively. As a reasoning-focused model, `o3-mini` consumes more tokens, increasing cost but often yielding more logically sound, well-structured rewrites for complex SQL transformations. Auxiliary LLM steps (e.g., grouping NLR2s and selecting the bottleneck) add negligible overhead. Overall, total cost scales approximately linearly with the number of candidate rewrites generated by the pipeline.

stage	time (s)			monetary cost ($)		
	suggest	semantic corr.	syntax corr.	suggest	semantic corr.	syntax corr.
o3-mini	5.8 (16.0%)	20.8 (57.6%)	9.6 (26.5%)	7.8e-03 (26.4%)	1.5e-02 (52.2%)	6.3e-03 (21.3%)
gpt-4o	1.5 (5.8%)	22.1 (87.9%)	1.6 (6.3%)	6.4e-03 (23.9%)	1.9e-02 (70.1%)	1.6e-03 (6.0%)
gpt-4o-mini	1.9 (2.6%)	66.8 (93.1%)	3.1 (4.3%)	4.0e-04 (11.2%)	3.1e-03 (84.6%)	1.5e-04 (4.2%)

Table 9. LLM cost analysis: the monetary and time cost of generating one rewrite candidate for a TPC-DS query, including all three stages— rewrite suggestions, semantic correction, and syntactic correction.

Table 9 shows that semantic correction dominates both time and cost across models. With `o3-mini`, more time also goes to candidate suggestion and syntax correction: its stronger reasoning yields more aggressive rewrites (more chances for big wins) but requires extra iterations to ensure equivalence. By contrast, `gpt-4o` and `gpt-4o-mini` rewrite more conservatively, needing fewer iterations but offering fewer opportunities for substantial speedups.

For equivalence checking, we set a 60 sec timeout for the tester and 5 sec for the verifier. Performance–evaluation time varied by query from seconds to hours. We also terminated execution early when a candidate failed to exceed the original query by the desired speedup threshold θ.

6 Discussion

Determining the equivalence of rewrites is a common challenge for all LLM-based approaches. Even the most powerful SQL verifier to date, SQLSolver [14], can verify only 18.9% of the generated TPC-DS rewrites. The verification rate is significantly higher for the JOB benchmark at 74.4%, yet it still fails to verify all rewrites. Moreover, as queries grow more complex, it becomes increasingly difficult for testers to spot inequivalent queries. To ensure correctness, all rewrites reported in the evaluation section have been manually inspected. Although verifying equivalence can be resource-intensive, the potential benefit of identifying a significantly more efficient rewrite often justifies the cost. The modest investment in LLM usage can yield substantial savings by reducing data warehouse compute costs, especially for long-running queries. In the context of recurring query execution, GenRewrite can serve as an automated system to propose query pairs that uncover valuable traditional rewrite rules. Consequently, GenRewrite is not designed to replace rule-based methods; rather, it leverages the capabilities of LLMs to identify promising rules that can be integrated into existing database. GenRewrite returns verified query pairs that deliver measured speedups together with the associated NLR2. A rule distilled from only one such pair is brittle and often fails to generalize—it overfits incidental syntax choices and misses necessary preconditions. We can aggregate multiple verified pairs tied to the same NLR2, and extract the common AST/plan transformation plus explicit guards/preconditions. As more pairs accumulate, the resulting pattern-matching rule becomes more general while remaining safe.

7 Related Work

In this section, we summarize the most relevant related work.

Query rewriting. Manual query rewriting does not scale and is prone to error. Much of the work in automated query rewriting is rule-based [9, 19, 24, 30, 33, 37]. LearnedRewrite [47] builds on Calcite's classic rewrite rules and explores how to find the optimal order of applying rules. More recent systems propose new rewrite rules tailored to analytical workloads [11, 22, 29, 35], as discussed in Section 5.2. The emergence of LLMs has inspired LLM-enhanced rule-based approaches. R-Bot [41] combines LLMs with evidence retrieval and self-reflection mechanisms to iteratively generate high-quality rewrites. LLM-R^2 [27] enhances a rule-based query rewriting system by using LLMs to suggest rewrite rules guided by contrastive demonstrations. Despite variations in the quality and quantity of their rules, rule-based techniques are fundamentally incapable of optimizing new query patterns that have not been seen before. Though synthesis-based approaches [15] are not constrained by rules, they struggle with complex queries, such as those in the TPC-DS benchmark, which require navigating an enormously large search space. To address these limitations, more recent work has explored rewriting beyond rule-based boundaries by leveraging LLMs more directly. LITHE [13] prompts an LLM with one selected natural language rule chosen from six handcrafted rewrite rules, each accompanied by an example, to guide the rewrite for a given query. Because this rule set is fixed, LITHE's knowledge cannot grow and evolve with more database feedback over time. QUITE [38] models the query rewrite process as a Markov Decision Process and at each timestep it chooses a refinement action (e.g., join reordering, predicate pushdown) by matching the query against a curated knowledge base built from database documentation and community sources. This design constrains the search to known actions and documented heuristics. In contrast, GenRewrite is not restricted to a fixed action library: it can propose novel rewrites and accumulate generalizable rewrite knowledge (NLR2s) over time, including patterns that are not present in existing literature.

LLMs in databases. Beyond query rewriting, LLMs support NL2SQL with strong Spider results [25]; schema matching by inferring column correspondences [32]; data augmentation with SQLsynth [42];

equivalence checking via LLM-SQL-Solver with pragmatic "relaxed" equivalence [46]; tuning/diagnostics via DB-BERT and D-Bot [43, 49]; and broader DBMS integration for optimization and indexing in DB-GPT [45]. Collectively, these works position LLMs as versatile assistants across the database stack.

8 Conclusion

This paper presents GenRewrite, a holistic system that leverages LLMs for query rewriting beyond rules. We introduce Natural Language Rewrite Rules (NLR2s) for transferring knowledge from rewriting one query to another, and a novel counterexample-guided technique that iteratively corrects the syntactic and semantic errors in the rewrites. Our empirical analysis demonstrates that GenRewrite optimize a wider range of complex queries and achieve faster speedup than all baselines on the standard TPC-DS and JOB benchmarks and their SQLStorm-generated variants.

Acknowledgments

This work was supported in part by research gifts from Google. We also would like to thank the Google Cloud Research Credits Program for providing the cloud credits used to conduct our experiments.

References

[1] dbt Labs | Transform Data in Your Warehouse. https://www.getdbt.com/.
[2] LeetCode. https://leetcode.com/problemset/database/.
[3] Looker business intelligence platform embedded analytics. https://cloud.google.com/looker.
[4] OLAPBench. https://https://github.com/SQL-Storm/OLAPBench/.
[5] OpenAI Introduction. https://platform.openai.com/docs/introduction/tokens.
[6] Prompt Engineering for Generative AI. https://developers.google.com/machine-learning/resources/prompt-eng.
[7] TPC-DS Benchmark. https://www.tpc.org/tpcds/.
[8] TPC-H Benchmark. https://www.tpc.org/tpch/.
[9] Rafi Ahmed, Allison Lee, Andrew Witkowski, Dinesh Das, Hong Su, Mohamed Zait, and Thierry Cruanes. 2006. Cost-based query transformation in Oracle. In *VLDB*, Vol. 6. 1026–1036.
[10] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.
[11] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. 2022. Computation reuse via fusion in Amazon Athena. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1610–1620.
[12] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, et al. 2016. The snowflake elastic data warehouse. In *Proceedings of the 2016 International Conference on Management of Data*. 215–226.
[13] Sriram Dharwada, Himanshu Devrani, Jayant Haritsa, and Harish Doraiswamy. 2025. Query rewriting via llms. *arXiv preprint arXiv:2502.12918* (2025).
[14] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving Query Equivalence Using Linear Integer Arithmetic. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.
[15] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. 2023. SlabCity: Whole-Query Optimization Using Program Synthesis. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3151–3164.
[16] Sérgio Fernandes and Jorge Bernardino. 2015. What is bigquery?. In *Proceedings of the 19th International Database Engineering & Applications Symposium*. 202–203.
[17] Han Fu, Chang Liu, Bin Wu, Feifei Li, Jian Tan, and Jianling Sun. 2023. CatSQL: Towards Real World Natural Language to SQL Applications. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1534–1547.
[18] Dawei Gao, Haibin Wang, Yaliang Li, Xiuyu Sun, Yichen Qian, Bolin Ding, and Jingren Zhou. 2023. Text-to-SQL Empowered by Large Language Models: A Benchmark Evaluation. CoRR abs/2308.15363 (2023). *arXiv preprint arXiv:2308.15363* (2023).
[19] Goetz Graefe. 1987. *Rule-based query optimization in extensible database systems.* Technical Report. University of Wisconsin-Madison Department of Computer Sciences.

[20] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.
[21] George Katsogiannis-Meimarakis and Georgia Koutrika. 2023. A survey on deep learning approaches for text-to-SQL. *The VLDB Journal* (2023), 1–32.
[22] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment* 13, 3 (2019), 348–361.
[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[24] Alon Yitzchak Levy and Inderpal Singh Mumick. 1997. Query optimization by predicate move-around. US Patent 5,659,725.
[25] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. 2023. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems* 36 (2023), 42330–42357.
[26] Peng Li, Yeye He, Dror Yashar, Weiwei Cui, Song Ge, Haidong Zhang, Danielle Rifinski Fainman, Dongmei Zhang, and Surajit Chaudhuri. 2023. Table-gpt: Table-tuned gpt for diverse table tasks. *arXiv preprint arXiv:2310.09263* (2023).
[27] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. *arXiv preprint arXiv:2404.12872* (2024).
[28] Weizheng Lu, Jiaming Zhang, Jing Zhang, and Yueguo Chen. 2024. Large Language Model for Table Processing: A Survey. *arXiv preprint arXiv:2402.05121* (2024).
[29] Abhishek Modi, Kaushik Rajan, Srinivas Thimmaiah, Prakhar Jain, Swinky Mann, Ayushi Agarwal, Ajith Shetty, Shahid K I, Ashit Gosalia, and Partho Sarthi. 2021. New query optimization techniques in the spark engine of azure synapse. *Proceedings of the VLDB Endowment* 15, 4 (2021), 936–948.
[30] M Muralikrishna et al. 1992. Improved unnesting algorithms for join aggregate SQL queries. In *VLDB*, Vol. 92. Citeseer, 91–102.
[31] Avanika Narayan, Ines Chami, Laurel Orr, Simran Arora, and Christopher Ré. 2022. Can foundation models wrangle your data? *arXiv preprint arXiv:2205.09911* (2022).
[32] Marcel Parciak, Brecht Vandevoort, Frank Neven, Liesbet M Peeters, and Stijn Vansummeren. [n. d.]. Schema matching with large language models: an experimental study (2024). *URL https://arxiv. org/abs/2407.11852* ([n. d.]).
[33] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record* 21, 2 (1992), 39–48.
[34] Mohammadreza Pourreza and Davood Rafiei. 2024. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in Neural Information Processing Systems* 36 (2024).
[35] Partho Sarthi, Kaushik Rajan, Akash Lal, Abhishek Modi, Prakhar Jain, Mo Liu, Ashit Gosalia, and Saurabh Kalikar. 2020. Generalized {Sub-Query} Fusion for Eliminating Redundant {I/O} from {Big-Data} Queries. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 209–224.
[36] Tobias Schmidt, Viktor Leis, Peter Boncz, and Thomas Neumann. 2025. SQLStorm: Taking Database Benchmarking into the LLM Era. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4144–4157.
[37] Praveen Seshadri, Joseph M Hellerstein, Hamid Pirahesh, TY Cliff Leung, Raghu Ramakrishnan, Divesh Srivastava, Peter J Stuckey, and S Sudarshan. 1996. Cost-based optimization for magic: Algebra and implementation. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. 435–446.
[38] Yuyang Song, Hanxu Yan, Jiale Lao, Yibo Wang, Yufei Li, Yuanchun Zhou, Jianguo Wang, and Mingjie Tang. 2025. QUITE: A Query Rewrite System Beyond Rules with LLM Agents. *arXiv preprint arXiv:2506.07675* (2025).
[39] Yoshihiko Suhara, Jinfeng Li, Yuliang Li, Dan Zhang, Çağatay Demiralp, Chen Chen, and Wang-Chiew Tan. 2022. Annotating columns with pre-trained language models. In *Proceedings of the 2022 International Conference on Management of Data*. 1493–1503.
[40] Ruoxi Sun, Sercan Ö Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. [n. d.]. Sql-palm: Improved large language model adaptation for text-to-sql. CoRR, abs/2306.00739, 2023a. doi: 10.48550. *arXiv preprint ARXIV.2306.00739* ([n. d.]).
[41] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2024. R-bot: An llm-based query rewrite system. *arXiv preprint arXiv:2412.01661* (2024).
[42] Yuan Tian, Daniel Lee, Fei Wu, Tung Mai, Kun Qian, Siddhartha Sahai, Tianyi Zhang, and Yunyao Li. 2025. Text-to-SQL Domain Adaptation via Human-LLM Collaborative Data Annotation. In *Proceedings of the 30th International Conference on Intelligent User Interfaces*. 1398–1425.
[43] Immanuel Trummer. 2022. DB-BERT: a Database Tuning Tool that" Reads the Manual". In *Proceedings of the 2022 international conference on management of data*. 190–203.
[44] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International*

Conference on Management of Data. 94–107.
[45] Guoliang Li Xuanhe Zhou, Zhaoyan Sun. 2023. DB-GPT: Large Language Model Meets Database.
[46] Fuheng Zhao, Lawrence Lim, Ishtiyaque Ahmad, Divyakant Agrawal, and Amr El Abbadi. 2023. Llm-sql-solver: Can llms determine SQL equivalence? *arXiv preprint arXiv:2312.10321* (2023).
[47] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (2021), 46–58.
[48] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-bot: Database diagnosis system using large language models. *arXiv preprint arXiv:2312.01454* (2023).
[49] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2024. D-Bot: Database Diagnosis System using Large Language Models. *Proceedings of the VLDB Endowment* 17, 10 (2024), 2514–2527.

Received July 2025; revised October 2025; accepted November 2025