R-Bot: An LLM-based Query Rewrite System

Zhaoyan Sun
Tsinghua University
szy22@mails.tsinghua.edu.cn

Xuanhe Zhou
Shanghai Jiao Tong University
zhouxh@cs.sjtu.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Xiang Yu
Huawei Company
yuxiang44@huawei.com

Jianhua Feng
Tsinghua University
fengjh@tsinghua.edu.cn

Yong Zhang
Tsinghua University
zhangyong05@tsinghua.edu.cn

ABSTRACT

Query rewrite is essential for optimizing SQL queries to improve their execution efficiency without changing their results. Traditionally, this task has been tackled through heuristic and learning-based methods, each with its limitations in terms of inferior quality and low robustness. Recent advancements in LLMs offer a new paradigm by leveraging their superior natural language and code comprehension abilities. Despite their potential, directly applying LLMs like GPT-4 has faced challenges due to problems such as hallucinations, where the model might generate inaccurate or irrelevant results. To address this, we propose *R-Bot*, an LLM-based query rewrite system with a systematic approach. We first design a multi-source rewrite evidence preparation pipeline to generate query rewrite evidences for guiding LLMs to avoid hallucinations. We then propose a hybrid structure-semantics retrieval method that combines structural and semantic analysis to retrieve the most relevant rewrite evidences for effectively answering an online query. We next propose a step-by-step LLM rewrite method that iteratively leverages the retrieved evidences to select and arrange rewrite rules with self-reflection. We conduct comprehensive experiments on real-world datasets and widely used benchmarks, and demonstrate the superior performance of our system, *R-Bot*, surpassing state-of-the-art query rewrite methods. The *R-Bot* system has been deployed at Huawei and with real customers, and the results show that the proposed *R-Bot* system achieves lower query latency.

PVLDB Reference Format:
Zhaoyan Sun, Xuanhe Zhou, Guoliang Li, Xiang Yu, Jianhua Feng, and Yong Zhang. R-Bot: An LLM-based Query Rewrite System. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/curtis-sun/LLM4Rewrite.

1 INTRODUCTION

Query rewrite is designed to transform an SQL query into a logically equivalent version that is more efficient to execute, playing a crucial role in enhancing query performance in numerous practical scenarios. In Huawei's real-world database deployments, slow queries are regularly identified and optimized for improved performance. For instance, while migrating an enterprise's core application, we rewrote 20 critical queries, leading to a 3.7x reduction in workload latency. Despite its significance, the process of query rewrite is NP-hard [61, 63], meaning there is a vast collection of possible rewrite rules, and the number of potential rule combinations increases exponentially. This complexity makes identifying an effective combination of rules a challenging and laborious task. There are two main paradigms for addressing this challenge.

Table 1: *R-Bot* v.s. Existing Query Rewrite Methods.

Method	Heuristic Fixed Order	Heuristic Exploring	Traditional Learning	*R-Bot* LLM&Evidence
Train	No	No	*Yes*	No
#Rules	High	*Low*	High	High
Quality	*Low*	*Low*	High	High
Robust	High	High	*Low*	High

Heuristic-based Methods. Some heuristic-based methods apply the rules in a fixed order derived from practical experience (e.g., PostgreSQL [10]). However, they may not achieve optimal results for queries requiring different rule orders, thereby risking the omission of essential rewrite sequences. Furthermore, other heuristic-based methods (e.g., Volcano [25]) attempt to comprehensively explore various rule orders through heuristic acceleration. Nevertheless, they might overlook dependencies among rules, such as an initially inapplicable rule could be activated by another, leading to the potential neglect of vital rewrite sequences. Thus, heuristic-based approaches are often criticized for their *inferior quality*.

Learning-based Methods. To optimize query rewrite, learning-based methods have been proposed [61, 63]. These methods employ neural networks that are trained on historical query rewrites to identify and apply the most advantageous rules for rewriting a query. However, learning-based approaches face criticism for their *low robustness*. For example, models trained through these methods struggle to adapt to unseen database schemas without undergoing additional training on new query rewrite examples (e.g., hundreds of examples), which may not be readily available in real scenarios.

Recent advances in large language models (LLMs) have shown superiority in understanding natural language and code, as well as reasoning ability [13, 21, 22, 26, 28, 30, 33, 39, 43, 47, 48, 56, 58, 59]. As LLMs can capture the query rewrite capabilities by pre-training from database forums and codes, encompassing both the direct rewriting ability of applying a rule and the indirect rewriting ability to draw inspiration from multiple dependent rules, we can leverage

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

LLMs to guide the query rewrite, particularly for slow queries that often remain as bottlenecks.

To realize this target, we aim to develop an *LLM-based query rewrite system* with three main advantages: *(1) High Quality.* On one hand, our system can figure out the potential rules that have implicit relations with the query (e.g., ones that become applicable only after applying other rules). On the other hand, our system can understand the interrelations among rules and generate an effective sequence of rules for holistic improvements. *(2) Zero-Shot Robustness.* Unlike conventional learning-based approaches that are limited to in-distribution data [61, 63], our system harnesses LLMs, whose extensive pre-training empowers it to adapt to new datasets seamlessly without the need for additional retraining. *(3) Executability and Equivalence.* Our system ensures the rewritten query is both executable and functionally equivalent to the original one, as it performs query rewrite by selecting and ordering well-crafted rewrite rules from established query optimization engines, rather than directly rewriting the queries via LLMs.

However, directly utilizing LLMs for query rewrite proves to be ineffective due to their tendency for hallucination [27]. For example, despite being pre-trained on a vast corpus of query rewrite data (e.g., Stack Overflow [12]), employing the advanced LLM GPT-4 to directly rewrite queries in DSB benchmark [18] yielded only a 5.3% success rate, which is significantly low. This highlights two primary challenges associated with leveraging LLM for query rewrite.

C1: How to mitigate LLM's factuality hallucination in query rewrite? It's common for LLM to encounter confusion during query rewrite, suggesting an intuitive approach of guiding LLM with specific rewrite evidences (e.g., database Q&As, database manuals and codes, forum, etc) closely related to the query. However, several challenges arise from this approach. Firstly, LLM often struggles to interpret the raw rewrite evidence due to difficulties in aggregating fragmented knowledge across various rewrite documents and understanding complex query rewrite codes. Secondly, it's crucial to sift through and identify the most beneficial rewrite evidences to serve as references for LLM, thereby steering it towards a more efficient rule selection for query rewrite.

C2: How to mitigate LLM's faithfulness hallucination in query rewrite? LLM encounters challenges in accurately analyzing complex queries, such as those containing multiple sub-queries, and in fully leveraging detailed rewrite evidences. Thus, it becomes essential to develop a multi-step LLM rewrite method that breaks down the query rewrite process into more manageable segments, thereby aligning with LLM's capabilities for better performance.

To tackle the above challenges, we propose *R-Bot*, an LLM-based query rewrite system designed with a systematic approach. First, we gather and prepare rewrite evidences from diverse sources, including integrated well-formatted rewrite rules aggregating from rewrite documents and summarizing from the complex rule codes, as well as high-quality Q&As from database forums (addressing **C1**). Second, for an input SQL query, we propose a hybrid structure-semantics method for retrieving pertinent evidences, including rewrite rules by matching functions and rewrite Q&As with both query structure and rewrite semantics similarities (addressing **C1**). To enable LLM to comprehend the rewrite evidences, we synthesize this information into *rewrite recipes*, which detail in natural language how to utilize the Q&As and rewrite rules for rewriting an SQL query, and we retrieve most relevant rewrite recipes to guide LLMs for query rewrite (addressing **C2**). Third, we design a step-by-step LLM rewrite algorithm, which guides LLM to iteratively utilize the rewrite recipes to refine its rewrite rule selection and ordering, possibly choosing a more promising rule sequence by reflecting and self-improving the rewrite process (addressing **C2**).

Contributions. In summary, we make the following contributions.
(1) We develop an LLM-based query rewrite system, *R-Bot*, which can select an effective rewrite rule sequence to guide query rewrite engines to rewrite a query (see Section 3).
(2) We design a multi-source rewrite evidence preparation pipeline, including clustering-based document reorganization and hierarchical code summarization for rewrite codes (see Section 4).
(3) We propose a hybrid structure-semantics retrieval method for retrieving relevant rewrite evidences (see Section 5).
(4) We propose a step-by-step LLM rewrite method that iteratively leverages the retrieved Q&As and rewrite recipes to select and arrange rewrite rules with self-reflection (see Section 6).
(5) Our experimental results on real-world datasets and widely used benchmarks demonstrate that *R-Bot* can significantly outperform existing state-of-the-art query rewrite methods. We have deployed *R-Bot* at Huawei with real customers, achieving much higher real-world query rewrite performance (see Section 7).

2 PRELIMINARIES

2.1 Query Rewrite

Rewriting a query involves numerous query transformations, and we typically identify and summarize common transformation patterns into rewrite rules. For instance, a query rewrite rule could involve replacing an outer join with an inner join when they are equivalent. The composition of these rules offers the flexibility to accommodate a wide range of query rewrite requirements.

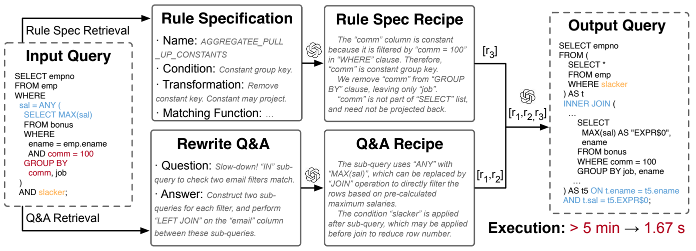
Definition 2.1 (Rewrite Rule). A rewrite rule r is denoted as a triplet (c, t, f), where c is the condition to use the rule, t is the transformation to be applied, and f is a matching function used for evaluation. If a query q satisfies the condition c as determined by the matching function f, then the transformation t can be applied to the query q, resulting an equivalent rewritten query $q^{[r]}$.

For example, Table 2 shows some query rewrite rules. Since the column "comm" with condition "comm=100" in the SQL query satisfies the condition "some 'GROUP BY' key is constant across rows" of rule r_3, we can apply the rule and remove the column "comm" from the "GROUP BY" clause.

Numerous rewrite rules have been pragmatically incorporated into database products, as evidenced in existing literature and products such as PostgreSQL [10], MySQL [9], Apache Calcite [15]. However, when applying the rules to rewrite a query, it is often cumbersome to decide the best rule sequence for two primary reasons. First, since a rewrite rule sometimes degrades the query (e.g., $q^{[r]}$ with higher execution latency than q), we should examine whether or not to use the rule. Second, it is also important to decide the order of applying the rules. For instance, applying one rule may render another rule obsolete. Thus query rewrite aims to find an optimal rule sequence to rewrite a query in order to minimize the execution cost of the rewritten query, which is formulated as below.

Table 2: Example Rewrite Rules.

	Rule	Condition	Transformation	Matching Function
r_1	FILTER_SUB_QUERY_TO_JOIN	Scalar, “IN”, or “EXISTS” sub-query in “WHERE” clause.	Transformed to join on the correlated column.	b -> b.operand(Filter.class) .predicate(containsSubQuery);
r_2	FILTER_INTO_JOIN	Filter condition with column from one join side.	Push down condition to filter non-nullable side.	b -> b.operand(Filter.class) .oneInput(Join.class);
r_3	AGGREGATE_PULL_UP_CONSTANTS	Some “GROUP BY” key is constant across rows.	Remove constant key. Constant may project.	b -> b.operand(Aggregate.class) .predicate(hasConstantExps);



Input Query

```
SELECT empno
FROM emp
WHERE
 sal = ANY (
  SELECT MAX(sal)
  FROM bonus
  WHERE
   ename = emp.ename
   AND comm = 100
  GROUP BY
   comm, job
 )
 AND slacker;
```

Rule Specification

· Name: *AGGREGATEE_PULL_UP_CONSTANTS*
· Condition: *Constant group key.*
· Transformation: *Remove constant key. Constant may project.*
· Matching Function: *...*

Rule Spec Recipe

The “comm” column is constant because it is filtered by “comm = 100” in “WHERE” clause. Therefore, “comm” is constant group key.
We remove “comm” from “GROUP BY” clause, leaving only “job”.
“comm” is not part of “SELECT” list, and need not be projected back.

Rewrite Q&A

· Question: *Slow-down! “IN” sub-query to check two email filters match.*
· Answer: *Construct two sub-queries for each filter, and perform “LEFT JOIN” on the “email” column between these sub-queries.*

Q&A Recipe

The sub-query uses “ANY” with “MAX(sal)”, which can be replaced by “JOIN” operation to directly filter the rows based on pre-calculated maximum salaries.
The condition “slacker” is applied after sub-query, which may be applied before join to reduce row number.

Output Query

```
SELECT empno
FROM (
  SELECT *
  FROM emp
  WHERE slacker
 ) AS t
 INNER JOIN (
  ...
    SELECT
     MAX(sal) AS "EXPR$0",
     ename
    FROM bonus
    WHERE comm = 100
    GROUP BY job, ename
   ...
 ) AS t5 ON t.ename = t5.ename
 AND t.sal = t5.EXPR$0;
```

Execution: > 5 min → 1.67 s

Figure 1: Query Rewrite Example: the sequence of rewrite rules $[r_1, r_2, r_3]$ is found by *R-Bot* based on retrieved rewrite rule specifications and rewrite Q&As.

Table 3: Rewrite Rule Specification v.s. Rewrite Rule.

	Rule Specification	Rewrite Rule
Mapping	Single/Multiple Rules	Single Rule
Source	Document/Code	Code in Single Engine
Clarity	LLM-Readable	Compiler-Readable
Selection	Match Function	LLM&Match Function

Definition 2.2 (Rule-based Query Rewrite). Consider a query q and a set of rewrite rules R. Assume α is a sequence of certain rewrite rules selected from R. Let q^{α} denote the rewritten query by sequentially applying the rules in α to rewrite q. Query rewrite aims to obtain an optimal rule sequence α^*, such that the execution cost of q^{α^*} is minimized among all possible rewritten queries of q.

The query rewrite problem has been proven to be NP-hard [61]. Traditional methods cannot select high-quality rules. To address this limitation, we advocate for utilizing LLM to select rewrite rules. To address the hallucination problem of LLMs [27], we perform an offline stage to extract query rewrite evidences, including rewrite Q&As and rewrite rule specifications, and store them as Q&A repository and rewrite rule specification repository respectively. During the online phase, given a SQL query, we retrieve relevant Q&As and rule specifications, and generate rewrite recipes, which outline how to rewrite a query using rewrite Q&As and rule specifications, assisting LLMs in comprehending the rewrite evidences. With the assistance of the rewrite recipe, LLMs are then guided through a step-by-step process to judiciously select and apply rewrite rules to the query. Next we formally define these notations.

Definition 2.3 (Rewrite Q&A). A rewrite Q&A includes a query rewrite question and a rewrite answer on how to rewrite the query in natural language.

Definition 2.4 (Rewrite Rule Specification). A rewrite rule specification is used to describe a rewrite rule (c, t, f) using natural language, which is also a triplet (nc, nt, f), where nc describes the condition and nt describes how to apply the transformation in natural language.

For example, Figure 1 shows a rewrite rule specification derived from Calcite code of rule “AGGREGATE_PULL_UP_CONSTANTS”, which explains the condition and transformation in natural language. Besides, the rule specification will be retrieved as evidence in this example, and we can use this evidence to rewrite the query.

Based on the general concepts, we further introduce definitions related to particular query rewrite, including rewrite Q&A recipe and rewrite rule specification recipe.

Definition 2.5 (Rewrite Q&A Recipe). Given a query q and a Q&A, a Q&A recipe provides instructions in natural language on how to utilize the Q&A to rewrite the query q.

Definition 2.6 (Rewrite Rule Specification Recipe). Given a query q and a rewrite rule specification, a rewrite rule specification recipe describes how to use the rewrite rule specification to rewrite the query q in natural language.

For instance, Figure 1 shows examples of rule specification recipe and Q&A recipe. Note that the rewrite Q&A may elaborate on application of multiple rules, and thus Q&A recipes can assist in guiding rule selection by considering the interrelations among these rules. Besides, rule specification recipes, derived from various sources like documents and codes, provide complementary insights to support query rewrite, as illustrated in Table 3.

In this paper, we focus on how to prepare the Q&A repository and rewrite rule specification repository (see Section 4), how to generate Q&A recipe and rule specification recipe (see Section 5), and how to use them to rewrite a query (see Section 6).

2.2 Large Language Models

Since LLMs have hallucinations [27], retrieval-augmented generation (RAG) has been proposed to mitigate this issue by indexing task-specific knowledge, retrieving relevant content for a given query, and generating answers based on the retrieved context [23, 35, 52, 53]. However, existing RAG techniques face limitations when applied directly to query rewrite for two primary reasons. First, there is absence of embedding methods capable of accurately evaluating the similarity between Q&As and the input query. For example, the conventional RAG technique relies on text embeddings for both the Q&A and the query, capturing only their semantic similarity while neglecting structural information pertinent to query rewrite. To address this gap, we propose a hybrid structure-semantics approach for retrieving relevant Q&As, as detailed in Section 5. Second, the multitude of rules presents a significant challenge for LLMs, as directly arranging these rules can lead to serious hallucination problems. This necessitates the development of a task-specific, step-by-step LLM algorithm that breaks down the query rewrite process into simpler, more manageable stages, as elaborated in Section 6.

3 THE OVERVIEW OF *R-BOT*

R-Bot includes an offline stage and an online stage (see Figure 2).

Offline Rewrite Evidence Preparation. This stage aims to extract rewrite Q&As from the Web and rewrite rule specifications from rewrite codes and database documents, subsequently storing

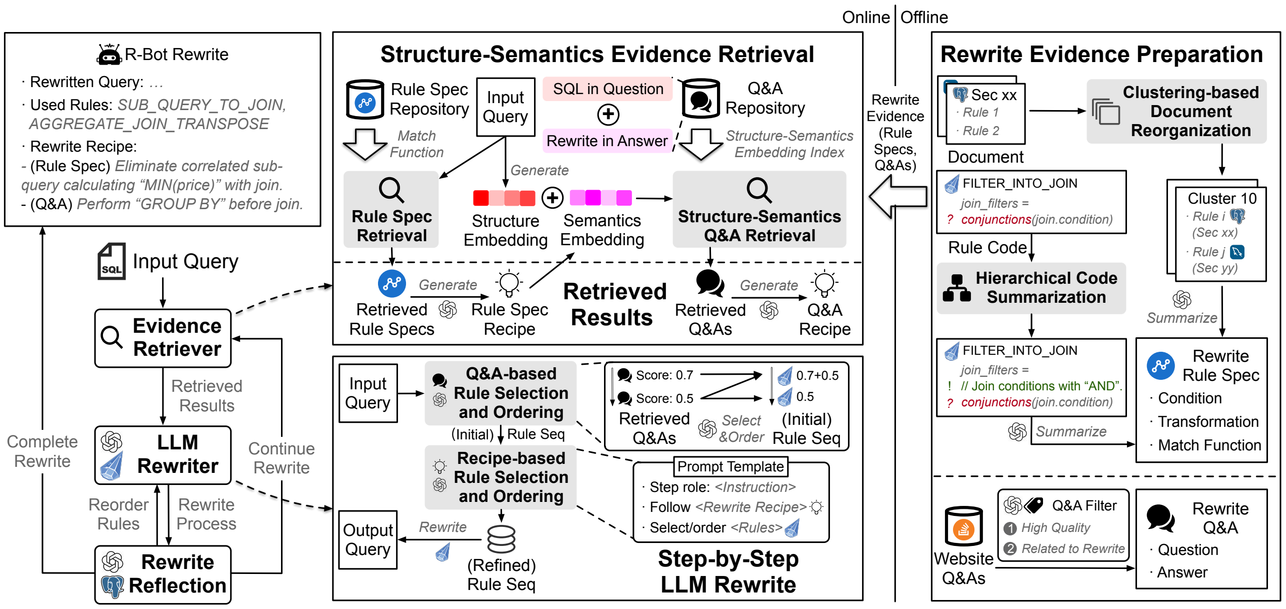


Figure 2: Overview of *R-Bot*.

them in Q&A repository and rule specification repository, respectively. Rewrite rule specifications are general and apply to multiple SQL queries, while rewrite Q&As are specific and pertain to individual queries. These resources assist in rewriting an online SQL query by guiding LLM-based selection of rewrite rules, without compromising the SQL equivalence inherently ensured by the rules.

We extract rewrite rule specifications from two types of resources. (*i*) Database menus and documents. Given that query rewrite evidence is often dispersed across various documents, sections, and paragraphs, it becomes necessary to aggregate these evidences from diverse sources through semantic clustering and distill them into a coherent rewrite rule; (*ii*) Rule codes. Considering the complexity of the code, characterized by its intricate nested calls, we employ a hierarchical strategy to streamline the code from simple to complex. This approach involves initially summarizing straightforward functions, followed by recursively summarizing more complex functions. In this process, symbol references are clarified using functions that have already been summarized. Ultimately, this method results in a concise summary of the overall rule code, effectively condensing the rule into a simplified format.

We extract Q&As from website Q&As (e.g., Stack Overflow [12]). Given the wide range of topics and variable quality of these website Q&As, we filter out high-quality Q&As related to query rewrite, by SQL selections (e.g., on question tags) and LLM filtering.

Besides, we construct efficient indexes to enhance the performance of Q&A and rule specification retrieval. We will discuss the detailed techniques in Section 4.

Online LLM-Guided Query Rewrite. Given an online SQL query, this stage retrieves pertinent Q&As and rule specifications, leverages them to rewrite the query, and offers reflections that not only complete the rewrite process but facilitate its further refinement. While LLM may have encountered data resembling rewrite evidences during pre-training, their sparse density often leads to hallucination in query rewrite. To address this, we guide LLM rewrite using the pertinent rewrite evidences prepared in a concise format.

(1) Structure-Semantics Evidence Retrieval. For an online SQL query, we retrieve relevant evidences from rewrite rule specification repository and Q&A repository.

(*i*) Rule specification retrieval. As the rule specification has matching conditions and matching functions, we can easily retrieve the relevant rule specifications whose matching functions are satisfied by the input SQL query;

(*ii*) Q&A retrieval. There are two types of Q&As potentially relevant to the SQL query. First, the SQL questioned in Q&A structurally matches the input SQL query. We require a structure-aware matching method to retrieve such Q&As. To this end, we propose a query structure embedding composed of (*i*) the query template embedded by pre-trained embedding, and (*ii*) one-hot embedding which represents the query's matched rule specifications to assess the structural similarities. Second, Q&A semantically matches the input SQL query, such as the rewrite explanations provided in the answer section of the Q&A. However, SQL usually has no natural language rewrite explanations. To this end, we can leverage the retrieved rule specification to identify relevant Q&As based on their semantics. We propose a semantics embedding method to embed the rule specification and Q&A, and then assess their similarities based on a similarity function (e.g., L^2-distance [3]). Lastly, to retrieve Q&As relevant to the input query both structurally and semantically, we merge the query's structural embedding with its semantic embedding into a unified representation. We then build an embedding index for Q&A repository offline. For an online query, we utilize this index to efficiently retrieve top-k relevant Q&As.

For retrieved rule specification and Q&A of the input SQL query, we leverage LLM to generate SQL-aware rule specification recipe and Q&A recipe that describe how to utilize them to rewrite this SQL query. We will discuss the technical details in Section 5.

The retrieved rule specification recipe and Q&A recipe will be used to reformulate the SQL query in the following steps.

(2) Step-by-Step LLM Rewrite. Given the rewrite rules supported by the query rewrite engine (e.g., Apache Calcite [2]), we direct LLM to select pertinent rules to rewrite. As there are many rules, if we directly instruct LLM to arrange a rule sequence, LLM can encounter serious hallucination problem. To mitigate this problem, we propose a step-by-step LLM rewrite method that decomposes rule-based query rewrite into several simpler steps.

(i) Q&A-based rule selection and ordering. Given the rewrite rules and a sorted sequence of Q&As retrieved from the previous step, ranked by the relevance score, we select and rank the rules. We first initialize the score of each rule as 0. Then for each pair of a rule and a Q&A, we use LLM to evaluate whether they are relevant, i.e., whether the rule is applicable in light of the Q&A. If applicable, we increase the score of the rule by the score of this Q&A. Then by enumerating all the pairs of rules and Q&As, we can get the final score of each rule and rank the rules based on the final score. Since the relevance can be evaluated by LLM offline, this step can be efficiently executed by algorithms.

(ii) Recipe-based rule selection and ordering. Building on the preliminary rule sequence established in the previous step, we utilize LLM to sift through and exclude any rules that do not align with the recipe. A straightforward way is to enumerate each pair of recipes returned by step (1) and rules returned by step ($2.i$), and ask LLM to evaluate their relevance and rank the rules. However, there are two limitations. First, it may overlook the rule relevance, since a recipe may encompass multiple rules. Second, since the recipe is generated from queries and cannot be evaluated offline, assessing each pair with LLM becomes costly. To address this issue, we propose a filtering method to efficiently select the rules.

(iii) Rule-based rewrite. Based on the selected rules, we input them to the query rewrite engine to rewrite the query.

We will discuss the technical details in Section 6.

(3) Rewrite Reflection. It provides rewrite reflections to either further refine the query or to finalize the rewrite process. It has two reflection resources. The first involves getting the cost of the rewritten query from databases, comparing it with the cost of the query prior to its rewrite, and returning *complete* if the cost of the rewritten query is smaller; *continue* otherwise. [1] The second involves asking LLMs to check whether or not all the rewrite recipes are realized by the query rewrite in this step, and returning *complete* if yes; *continue* otherwise. So there are four possible reflections.

(i) *complete, complete:* It finalizes the rewrite process and returns the rewritten query.

(ii) *complete, continue:* It further rewrites the query by jumping to step (1) with the previously rewritten query as the input. Specifically, it starts a new round of LLM-guided query rewrite, where new rewrite evidences can be retrieved and new rules can be selected based on the previously rewritten query.

(iii) *continue, complete:* It further refines the query by jumping to step ($2.ii$), focusing on reordering the existing set of rules. Specifically, besides the recipes and the rules, we further input the rules actually used in the previous rewrite process, and instruct LLM to prioritize the unused rules.

(iv) *continue, continue:* This approach integrates elements from branches ($3.ii$) and ($3.iii$). Initially, it defaults to branch ($3.iii$) unless this path is revisited excessively, surpassing a predefined threshold. Under these circumstances, given that branch ($3.iii$) has exhaustively explored the rules for the current query, the process transitions to branch ($3.ii$). This shift initiates a new cycle, aiming to further refine the query.

Deployment at Huawei. We have deployed *R-Bot* at Huawei database GaussDB [32, 34]. Initially, GaussDB identifies slow SQL queries, such as those with execution time exceeding one minute. Then, GaussDB utilizes *R-Bot* to rewrite the detected query. Next, if the version rewritten by *R-Bot* proves to be more efficient than the one rewritten by GaussDB, the system caches the query pattern using its domain-specific language (DSL). This pattern is integrated into GaussDB non-intrusively via a SQL-like plugin, which activates instantly without requiring database version updates or code changes, ensuring a seamless experience for front-end applications. During runtime, if a new query matches the cached pattern, GaussDB transparently invokes *R-Bot* to rewrite the query and apply the optimization. Furthermore, *R-Bot* has been widely used to address slow SQLs at Huawei. We have also validated *R-Bot* on a real-world dataset from China's largest bank (ICBC) (see Section 7.5). The results demonstrate that *R-Bot* effectively optimizes real queries, significantly improving latency of 14 critical slow queries and reducing overall latency from 9.23 hours to 4.37 hours.

4 REWRITE EVIDENCE PREPARATION

We discuss how to extract and standardize rewrite evidences from diverse rewrite sources. This evidence is crucial for crafting a comprehensive rewrite recipe that guides the LLM rewrite process. We explain respectively how to prepare rewrite rule specifications (see Section 4.1) and rewrite Q&As (see Section 4.2).

4.1 Rewrite Rule Specification Preparation

The rewrite rule specification clearly outlines, in natural language, the condition for use, the query transformation operations to be executed, and the matching function used for evaluation. It contains three key components. (i) "condition": a prerequisite that a query must fulfill to utilize the rule; (ii) "transformation": detailed steps for transforming the query into an equivalent form that is optimized for more efficient execution; and (iii) "matching function": an executable function that outputs '1' if the input query matches the rule; and '0' otherwise.

Generating rewrite rule specifications is challenging due to the considerable effort needed to distill and synthesize information from various sources into a concise format. This process includes summarizing extensive rewrite codes, which may span thousands of lines and feature complex structures, as discussed in Section 4.1.1. Additionally, it integrates crucial but scattered information from documents on rewrite rules, as outlined in Section 4.1.2.

4.1.1 Transforming Rule Code into Rule Specification. Given that some query rewrite engines, such as Apache Calcite [2], are not accompanied by comprehensive documentation, we are compelled to decipher the rewrite rules directly from the raw code. This process involves navigating through complex code structures that include intricate nested calls. To address this challenge, we introduce a hierarchical rule code summarization method, as illustrated in Figure 3. Our approach begins with the construction of a rule code structure tree, emanating from the rule's main function. In this tree, each node represents a symbol declaration (e.g., functions, variables, classes), while each edge denotes a symbol reference relationship.

[1]Database statistics play a crucial role in query optimization. However, LLM often struggles to directly understand complex database structures and intricate statistical data. To address this, we leverage the statistics indirectly through query costs, utilizing these costs to guide the LLM reflection mechanism.

Progressing through the structure, we methodically summarize the declaration code, moving from simple to more complex elements and clarifying symbol references using previously summarized symbols. With the summary of the root node at our disposal, we guide LLM to convert this into a standardized rewrite rule specification.

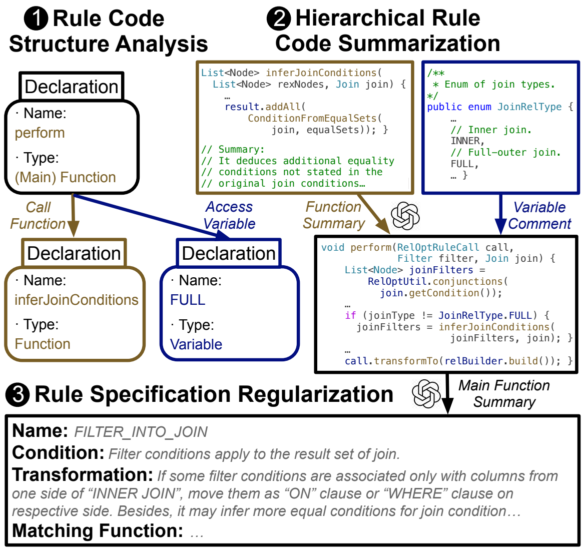
Rule Code Structure Analysis. Given the symbol references of the rule code, it is crucial to clarify the symbol declarations before summarizing the rule code. We first build a root node representing the main function. Then, we use code analysis tools (e.g., JavaSymbolSolver [1]) to associate the symbols with their corresponding declarations, which are children nodes of the root node. If the declaration of some nodes also accesses other unseen symbols, we further resolve its symbol references. We recursively expand the nodes until reaching built-in symbols. In this way, we obtain a rule code structure tree, where each node represents a symbol declaration and each edge represents a symbol reference relationship.

Hierarchical Rule Code Summarization. The complex rule code structure poses two challenges for LLM summarization. First, due to the relevance of nearly every declaration in the code, if we directly input them to LLM, the long context can greatly degrade LLM performance [31, 38]. Second, the substantial width and depth of the code structure (e.g., tens of nodes) further increase the reasoning burden. To address these issues, we propose a hierarchical rule code summarization method. First, if a node declaration already has detailed comments, summarization is unnecessary. Second, leaf node without comments can be directly summarized by LLM due to the simple code and absence of unfamiliar symbols. Third, for the non-leaf node whose children are already summarized, the symbol references in the declaration code can be clarified by its children summaries. Specifically, we insert the symbol summaries as comments into the declaration code, which enables accurate LLM summarization. This process is repeated recursively until the root node is summarized, yielding a summary of the entire rule code.

Rule Specification Regularization. To facilitate LLM understanding, we regularize the rule code as a standard rewrite rule specification. Specifically, we use LLM to extract the condition and transformation of the rule, using the prompt p_{reg} =*"Given a rewrite rule code summary, your task is to extract the rewrite rule that explains completely and detailedly the condition and transformation."*.

4.1.2 Transforming Rewrite Document into Rule Specification. Considering the variety of rewrite documents, such as those for PostgreSQL and MySQL, note that sections within a single document may cover rewrite rules that bear weak relation to one another. For example, optimizations for the "WHERE" clause might discuss both constant folding and index utilization without clear interrelation. Additionally, components complementary to a rule can be dispersed across different documents. For instance, while a MySQL document might detail conditions conducive to acceleration via index utilization, a separate PostgreSQL document could highlight how certain column transformations might inhibit the use of indexes. Together, these insights from disparate sources can contribute to forming a comprehensive rewrite rule specification.

To address this, we propose a clustering-based document reorganization method. First, we use LLM to extract rewrite rules from rewrite documents. Second, for extracted rules, we cluster the correlated ones together into one group, where we evaluate their semantics similarities by their text embeddings (e.g., SBERT [41]). Third, we use LLM to summarize each rule cluster, and transform each cluster summary as a regularized rewrite rule specification.



```
List<Node> inferJoinConditions(
  List<Node> rexNodes, Join join) {
    …
    result.addAll(
        ConditionFromEqualSets(
            join, equalSets)); }

// Summary:
// It deduces additional equality
// conditions not stated in the
// original join conditions…
```

```
/**
 * Enum of join types.
*/
public enum JoinRelType {
    …
    // Inner join.
    INNER,
    // Full-outer join.
    FULL,
    … }
```

```
void perform(RelOptRuleCall call,
             Filter filter, Join join) {
    List<Node> joinFilters =
        RelOptUtil.conjunctions(
          join.getCondition());
    …
    if (joinType != JoinRelType.FULL) {
      joinFilters = inferJoinConditions(
                     joinFilters, join); }
    …
    call.transformTo(relBuilder.build()); }
```

Name: *FILTER_INTO_JOIN*
Condition: *Filter conditions apply to the result set of join.*
Transformation: *If some filter conditions are associated only with columns from one side of "INNER JOIN", move them as "ON" clause or "WHERE" clause on respective side. Besides, it may infer more equal conditions for join condition…*
Matching Function: *…*

Figure 3: Transforming Rule Code into Rule Specification.

Rule Extraction. We use LLM to extract rewrite rules from the rewrite documents in two steps. First, if we directly input all the documents to LLM, it often overlooks important details in the middle of the extremely long context (e.g., 100k) [31, 38]. We thus split the documents into structured blocks (e.g., sections, sub-sections) that each can be effectively processed by LLM. Second, we instruct LLM to extract rewrite rules from the split blocks. To mitigate the hallucination problem [27], we require LLM to locate supporting content in the source document. If no such content can be found, we can deem the extraction low quality and repeat LLM extraction.

Rule Clustering. To identify pertinent rules among documents (e.g., condition push-down involved in both PostgreSQL and MySQL documents), we cluster the rules based on rule semantics. Specifically, we embed each rule into vectors (e.g., using multi-qa-mpnet-base-cos-v1 [11, 41]), and cluster them using Gaussian Mixture Models [42]. To enhance clustering in high-dimensional space, we apply Uniform Manifold Approximation and Projection for dimensionality reduction by approximating local data manifold [40].

Rule Specification Regularization. We use LLM to obtain standardized rule specifications from rule clusters. First, we summarize each cluster using LLM with prompt p_{summ} =*"Given rewrite rule components, your task is to summarize them into one paragraph, and your summary should include as many details as possible."* Then, we use LLM to transform the summary to regularized rewrite rule specification, by extracting rule condition and transformation.

4.2 Rewrite Q&A Preparation

The rewrite Q&A showcases how to rewrite a particular SQL query to an optimized one, which is composed of two parts: (i) "question" that denotes a query rewrite request; (ii) "answer" that provides the query transformations for the question. There are plenty of rewrite Q&As within online database community forums (e.g., millions at Stack Overflow [12]), often covering practical cases beyond standard rewrite rules. To filter high-quality rewrite Q&As from

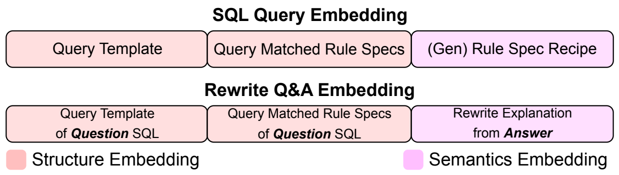


Figure 4: Structure-Semantics Embeddings in Q&A Retrieval.

these mixed sources, we use a hybrid method: first selecting by question tags (e.g., "query-optimization") and community feedback (e.g., Stack Overflow score higher than 3), and then using LLM to verify their relevance to query rewrite.

5 STRUCTURE-SEMANTICS RETRIEVAL

Considering the existence of vast repositories of rule specifications and Q&As, only a minimal fraction of these resources is pertinent to an online SQL query. Thus, there is a need to identify the relevant ones both effectively and efficiently. In this section, we introduce how to retrieve relevant rewrite evidences (including rule specifications and Q&As). First, we retrieve relevant rule specifications using a function-based rule retrieval method (see Section 5.1). Second, we retrieve relevant rewrite Q&As using a hybrid structure-semantics method, with both query structures and rewrite semantics aligned with the input query (see Section 5.2). Lastly, we generate tailored rewrite recipes for the input query by leveraging both the retrieved rule specifications and Q&As (see Section 5.3).

5.1 Rewrite Rule Specification Retrieval

Given an input SQL query q, we examine each rule specification and apply its associated matching function to get a boolean value which indicates whether the condition of the rule is satisfied. If the result is true, the corresponding rule specification rs is retrieved. Then, we use LLM to generate a rule specification recipe, using a prompt $p_{rule\_spec\_recipe}$ =*"Given an SQL query 'q' and a rewrite rule specification 'rs', your task is to explain concisely and detailedly how the rule applies to the query, by specifying (1) the SQL segments matched by the condition, and (2) the transformation of the rule."*

5.2 Rewrite Q&A Retrieval

There are two types of Q&As that could potentially enhance the query rewrite. Firstly, the questions within Q&As exhibit a high structural similarity to the input query, indicating that the answers of Q&As have the potential to assist in rewriting the query. Second, the answers within Q&As demonstrate a high semantic similarity to the input query. This suggests that the Q&As are capable of addressing similar issues or bottlenecks present in the input query, such as those involving a sub-query with an aggregate function.

Structure-Semantics Embeddings. To effectively identify these two types of Q&As, we propose structure-semantics embeddings for both SQL queries and Q&As (see Figure 4). (1) *SQL Query Embedding.* First, we introduce a structure-aware query embedding strategy, including (i) generating query templates embedded by pre-trained embedding (e.g., text-embedding-3-small [8]), and (ii) generating a one-hot embedding to indicate which rule specifications match the SQL query. The two embeddings capture the essential structural features to query rewrite (see Section 5.2.1). Second, we propose a semantic matching method designed to discern the semantic similarity between the SQL query and Q&As. However SQL may not encapsulate sufficient semantic information. Fortunately, the retrieved rule specification recipe of SQL in Section 5.1 encompasses these semantics, enabling us to derive an embedding from it. Third, we concatenate the structural and semantic embeddings to obtain a combined structure-semantics embedding. Given the possibility of retrieving multiple rule specification recipes, each SQL query has the potential to generate multiple embeddings. (2) *Rewrite Q&A Embedding.* Similarly, we generate embeddings of Q&As. For each Q&A, its embedding includes a structure-aware embedding of its query in the question part and a semantic embedding of its answer part. These embeddings allow us to identify Q&As that exhibit a high similarity to the input query in terms of their embeddings.

Structure-Semantics Q&A Retrieval. To optimize performance for structure-semantics Q&A retrieval, a unified structure-semantics embedding index (e.g., HNWS [3]) is constructed for Q&As offline. Given an SQL query, we generate its structure-semantics embedding and retrieve relevant Q&As using the index (see Section 5.2.2).

5.2.1 Structure-Aware SQL Query Embedding. Existing query embedding models [8, 46, 54, 55] cannot be directly applied for embedding query structure for query rewrite due to two key limitations. On one hand, the query-specific embeddings mostly adopt small-scale neural networks (e.g., lower than 0.1B) fine-tuned on narrow datasets [54, 55] or workloads [46], limiting their generalizability. On the other hand, the general text embeddings (e.g., text-embedding-3-small [8]) fail to effectively represent query structure, as they are sensitive to irrelevant information: (i) The identifiers (e.g., table/column names) and literals in the query often disrupt characterization of query structure. For example, while renaming the schema in a query can maintain the integrity of the query rewrites, this action results in a version with significantly altered semantics and it generates an embedding distinctly different from that of the original query; (ii) Since query rewrites typically focus on transforming only certain parts of the query (e.g., a sub-query), disregarding other irrelevant parts can help clarify the query structure; (iii) The text embedding models cannot guarantee commutative invariance, implying they may generate dissimilar embeddings for expressions such as "p_1 *AND* p_2" and "p_2 *AND* p_1". These can be addressed through carefully designed textual transformations.

To address the aforementioned challenges, we propose a structure-aware query embedding method, where each SQL query embedding is composed of two parts. The first part distills the core query structure as query templates, and uses pre-trained text embedding to embed them. The second part employs a one-hot encoding strategy to indicate whether the query matches a rule specification, with '1' representing a match and '0' indicating no match. The width of this embedding corresponds to the total number of rule specifications. When comparing the two query structure embeddings, the first one captures a more global representation of the query structure, whereas the second focuses on more local expressions.

Dataflow Based Query Template Embedding. To capture the essential SQL features for query rewrite, we initially reformulate the SQL query into composite dataflows that detail operations on the involved identifiers. Specifically, given a query q, a dataflow δ is a list of SQL operations sequentially performed on one or multiple tables or columns in the query, which corresponds to an SQL segment of the query, e.g., logical expression, mathematical expression, "WHERE" clause, "GROUP BY" clause, "FROM" clause, "JOIN" clause, "SELECT" list, sub-query, etc. For instance, we show sample dataflows for the input query in Figure 1, concentrating on the column "*bonus.sal*": $\{sal, SELECT\ MAX(sal), emp.sal = ANY\ (SELECT\ MAX(sal)\ ...)\}$ where "*sal*" is first input from "*bonus*" table, projected with an aggregate function "$MAX(\cdot)$", and then compared with the column "*emp.sal*" from the outer query.

Since we focus on logical query rewrite, dataflows can be considered independent if they do not involve common identifiers. Then, if we examine the rewrite rule from dataflow perspective, the condition must restrict the pattern of certain dataflows associated with a specific identifier. Otherwise, there are scarcely any potential query rewrites within a set of independent dataflows. Consequently, we can generate query templates that concentrate on the SQL operations related to each particular identifier.

To ensure query templates are robust to schema renaming, we propose an isolated masking method. If a query template involves multiple identifiers, swapping any two identifiers alters its semantics and disrupts the embedding. Thus, we derive multiple query templates from the query, with each template preserving only one identifier while masking the others. We then refine the templates in three steps. First, to limit the number of templates derived from complex queries, we retain only those where the SQL operations associated with the identifier are likely to match certain rewrite rules. Second, we simplify the query template by eliminating SQL operations that do not involve the key identifier. Third, for the commutative SQL operands, we sort them lexically, thereby rendering the template invariant to operand order.

Step 1: Potential Identifier Selection. Considering the SQL operations associated with the identifier, we observe that multiple appearances of the same identifier in mutually exclusive dataflows may prompt query rewrites by leveraging operation correlations. For instance, the twice appearance of column "a" in "$a < b$" and "$a = 5$" can indicate constant folding, replacing "$a < b$" with "$5 < b$". Then, given an SQL query, we propose two ways to measure identifier frequency: (i) Column appearances: By traversing the query, we count how often each column appears. Note that columns appearing as direct projections (e.g., "SELECT a") are not counted, as this does not involve any meaningful SQL operations. (ii) Table appearances: We count the appearances of the table which is utilized either in the "FROM" clause or within a "JOIN" operation. With the frequency of tables and columns, we focus on identifiers that appear more than once within the query. We then select their corresponding query templates as representative, ensuring we capture the most significant patterns for query rewrite.

Step 2: Query Template Reduction. Given the single identifier preserved in the query template, we standardize it as "$table$" for table and "*column*" for column. Moreover, to remove irrelevant query information, we mask other identifiers with "_" and the literals with "?". We make a distinction between identifier mask and literal mask, since they indicate different query rewrite potentials. For instance, we can preserve the constant folding pattern in the query template with "$column < \_\ AND\ column =?$", which means the constant equality can be transferred to other conditions involving "*column*". Then, we further simplify the query template following three steps. (i) If any dataflow in the query template involves only masked literals "?", no identifiers, and no non-deterministic functions, the dataflow can be regarded as a literal and replaced with literal mask "?". (ii) If any dataflow involves some identifiers but no explicit identifiers, we replace it with identifier mask "_", as it is independent to the potential identifier. (iii) For the dataflow of clause like "t_1 JOIN t_2 ON c", if the join condition "c" and a join table (e.g., "t_2") are both masked with "_", we replace the clause with "t_1" to remove irrelevant details. These steps are repeated to streamline the query template until it cannot be further matched, leaving a structure that retains only the essential core related to the potential identifier.

Step 3: Commutative Invariance Guarantee. We further transform the query template for commutative invariance. Specifically, we search for commutative operators in the query template (e.g., addition, set intersection), and sort their operands lexically (e.g., "$column < \_$" before "$column =?$"). Equipped with the refined query templates, we then obtain query template embeddings with pre-trained text embedding (e.g., text-embedding-3-small [8]).

One-Hot Embedding for Matched Rule Specifications. Given an SQL query, its matched rule specifications also reflect query structure. For instance, queries matching "SUB_QUERY_TO_JOIN" rule likely exhibit similar structures around sub-queries. As shown in Section 5.1, we first examine the rule specifications with their associated matching functions. Following this, we construct a one-hot encoding from the rule matching results, where each position corresponds to a rule specification. If a rule specification matches the query, its corresponding position is '1'; '0' otherwise. The one-hot embedding for matched rule specifications can capture local structural information indicated by the rule condition, complementary to the global structures captured by query template embedding.

5.2.2 *Structure-Semantics Q&A Retrieval.*

First, we build a structure-semantics embedding index for Q&A repository in three steps. (i) We extract the queries in the Q&As, generate query templates, and embed them with pre-trained text embedding (e.g., text-embedding-3-small [8]). We also match the queries with rule specifications to build one-hot embedding for matched rule specifications. (ii) We refine the Q&A by preserving the semantics of query rewrite and eliminating irrelevant text with the help of LLM, then embedding the streamlined information using a pre-trained text embedding model (e.g., text-embedding-3-small [8]). (iii) We normalize the three embeddings as unit vectors, concatenate them to form a holistic embedding, and insert them into the index. Next, for an online SQL query, we similarly generate its embedding and identify the most relevant Q&As with top-k embedding similarities.

Every SQL query may be associated with multiple templates, leading to multiple structural embeddings. Similarly, each SQL query encompasses multiple specification recipes, resulting in multiple semantic embeddings. We concatenate these embeddings for all possible combinations and, for each concatenated embedding, we utilize the index to identify top-k Q&As with the highest similarities. To combine the retrieved results from multiple embeddings, we adopt the Reciprocal Rank Fusion (RRF) method [17]. Specifically, for each retrieved list corresponding to an embedding, we assign a score to the i-th Q&A in the ranked list as, $s(qa_i) = \frac{1}{\alpha+i}$, where α is 60 by default. The RRF score of a Q&A across all the retrieved lists is calculated by summing its scores from each list (assigning a score of 0 if the Q&A is absent in a list). We then identify the Q&As with highest top-k RRF scores as the final selection, which comprehensively captures the structure and semantics relevance.

5.3 Rewrite Recipe Generation

For an input SQL query q, since the retrieved Q&A qa is used to rewrite similar but different queries, we first use LLM to generate rewrite recipes describing how to rewrite the input query inspired by the Q&A, using the prompt p_{qa_recipe} =*"Given an SQL query 'q' and a rewrite Q&A 'qa', your task is to propose some strategies on rewriting the query, by (1) transferring the Q&A strategy to the query, and (2) explaining the strategy detailedly."* Second, to integrate both rule specification recipes (see Section 5.1) and Q&A recipes, we condense those that are closely related and eliminate any duplicates. Thus, we utilize LLM to cluster them by their semantic similarity, and summarize each recipe cluster concisely into a single recipe.

6 STEP-BY-STEP LLM REWRITE

Given a rule-based query rewrite engine (e.g., Apache Calcite [2]), we utilize the retrieved Q&As and derived rewrite recipes to assist LLM in selecting appropriate rewrite rules from the engine to rewrite the input SQL query. Given that the number of possible rule sequences grows exponentially with the number of rules, LLMs are susceptible to errors when choosing from a vast array of rules. To address this challenge, we introduce a step-by-step filter-refinement method designed to meticulously select high-quality rules. Specifically, we start with a filtering step by employing an efficient Q&A-based method for rule selection and ordering, allowing us to preliminarily arrange the rules. Next, we instruct LLM to select the rules and arrange the order according to the recipes. Then, we feed the selected rules into the query rewrite engine to rewrite the query with the rules. Lastly, we evaluate the outcomes of the rewrite process to determine whether further refinement is needed or if the query rewrite can be considered complete.

Step 1: Q&A-based Rule Selection and Ordering. Given the rewrite rules and the input query, we filter the rules and arrange the order in three steps. First, we select the rules directly matched by the query, and assign them a relevance score according to their transformation to the query. Second, we select the rules indirectly relevant to the query using the retrieved Q&As. Based on the retrieval scores of the Q&As, we assign each indirectly relevant rule with a relevance score. Third, we rank the rules by their relevance scores in a descendent order, which serves as an initial order.

(i) We first filter rules whose matching functions are satisfied by the input query. However, the matched rules vary in relevance based on their actual transformation. Specifically, the rules of the query rewrite engine can be classified into two types of transformations: the normalization rule (e.g., "SUB_QUERY_TO_JOIN"), which nearly always reduces query cost; and the exploration rule (e.g., "AGGREGATE_JOIN_TRANSPOSE"), which transforms the query but does not consistently result in cost reduction [45]. We classify the rules based on expert experience, and assign relevance scores to the matched rules based on their transformation types. Initially, every rule has a score of $-\infty$. Then, the matched normalization rules are assigned $+\infty$ as closely relevant, and the matched exploration rules are assigned 0 as weakly relevant.

(ii) Besides the directly relevant rules, we also detect indirectly relevant rules using the retrieved Q&As. Specifically, for each pair of rule and Q&A, we leverage LLM to evaluate whether the rule can be applied in the context of the Q&A. If applicable, the rule's score is incremented by the Q&A similarity score to the input query. Note that if the rule has a score of $-\infty$, we first initialize it as 0 before increment. Enumerating all the pairs of rules and the retrieved Q&As, we obtain the final score of each rule r. *Note that the relevance can be performed by LLM offline for each possible pair of rule and Q&A, thus not affecting the algorithm's efficiency.*

(iii) Given the rules with relevance scores, we first filter out those with $-\infty$ as irrelevant. Then, we order the remaining rules by prioritizing the rules with higher scores, which comprehensively reflect their direct and indirect relevance to the input query.

Step 2: Recipe-based Rule Selection and Ordering. Derived recipes offer more detailed guidance for rewriting the input query compared to the basic rule specifications and Q&As. Therefore, we refine the initially arranged rule sequence to align it more closely with these recipes. This process is divided into three sub-steps.

(i) We first refine rule selection with recipes. A straightforward method is to assess the relevance between each rule and each recipe, excluding those rules deemed irrelevant. However, this method faces two significant drawbacks. Firstly, it overlooks rule dependencies, consequently excluding indirectly utilized rules that become relevant only after another rule is applied. Secondly, this method incurs high costs, especially since it lies on the critical path for online query rewrite. To address these limitations, we propose a batch filtering method. Specifically, we first feed the rewrite recipe and a batch of rules to LLM, and require LLM to select the most appropriate rules aligned with the rewrite recipe, using the prompt p_{select_rule} =*"Given the input query and rewrite rules, you should evaluate whether the rules can be applied to rewrite the query in the context of the recipe."* Next, we feed the previously selected rules along with the next batch of rules into LLM for further selection. We continue this process iteratively until all rules have been evaluated. In this way, we not only accelerate the LLM selection process, but also consider the rule dependency within the selected rules.

(ii) Next, we methodically refine the rule order according to rewrite recipe. First, taking into account the inter-dependence of the selected rules, we categorize them into groups wherein each group pertains to the same SQL operator (e.g., join). We then instruct LLM to arrange them following the rewrite recipe, emphasizing the alignment of closely related rules. Second, leveraging the grouped rule ordering as a basis, we instruct LLM to refine the overall rule ordering to optimally align with the rewrite recipe.

(iii) We input the refined rule sequence to the query rewrite engine, and rewrite the input query with the rules.

Step 3: Rewrite Reflection. With regards to the unstable performance of LLM in complex tasks like rule arrangement [50, 51], we reflect the rewrite process to determine whether to finalize or refine the rewrite as discussed in Section 3.

7 EXPERIMENTS

7.1 Experiment Setting

We implement our system *R-Bot* using the rules in an open-sourced query engine Apache Calcite [15]. We execute SQL queries in PostgreSQL v14 on a machine with 128 GB RAM and 3.1GHz CPU.

Datasets. To verify the effectiveness of *R-Bot* on different scenarios, we conduct experiments on three types of datasets. (i) TPC-H is a standard OLAP benchmark, which contains 62 columns and 44 queries. We separately test *R-Bot* on different data sizes, i.e.,

Table 4: Comparison of Query Latency.

Query Latency (s)	TPC-H 10x			DSB 10x			Calcite (uni)		
	Average	Median	p90	Average	Median	p90	Average	Median	p90
Origin	104.86	10.60	300.00	37.76	5.28	300.00	109.73	56.35	300.00
LearnedRewrite	69.60 (↓33.6%)	12.26 (↑15.7%)	300.00 (↓0.0%)	30.47 (↓19.3%)	5.28 (↓0.0%)	55.02 (↓81.7%)	79.07 (↓27.9%)	5.24 (↓90.7%)	300.00 (↓0.0%)
GPT-3.5	85.98 (↓18.0%)	10.60 (↓0.0%)	300.00 (↓0.0%)	37.75 (↓0.0%)	5.36 (↑1.5%)	300.00 (↓0.0%)	55.41 (↓49.5%)	22.74 (↓59.6%)	230.99 (↓23.0%)
GPT-4	67.10 (↓36.0%)	10.60 (↓0.0%)	300.00 (↓0.0%)	37.77 (↑0.0%)	4.92 (↓6.8%)	300.00 (↓0.0%)	60.86 (↓44.5%)	20.06 (↓64.4%)	300.00 (↓0.0%)
R-Bot (GPT-3.5)	**55.71 (↓46.9%)**	10.41 (↓1.8%)	300.00 (↓0.0%)	26.19 (↓30.6%)	4.61 (↓12.7%)	35.25 (↓88.2%)	37.71 (↓65.6%)	8.37 (↓85.1%)	65.67 (↓78.1%)
R-Bot (GPT-4)	57.60 (↓45.1%)	**10.37 (↓2.2%)**	300.00 (↓0.0%)	**25.35 (↓32.9%)**	**4.58 (↓13.2%)**	**17.17 (↓94.3%)**	**12.45 (↓88.6%)**	**5.04 (↓91.0%)**	**48.30 (↓83.9%)**

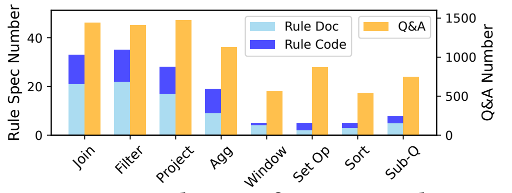


Figure 5: Distribution of Rewrite Evidences.

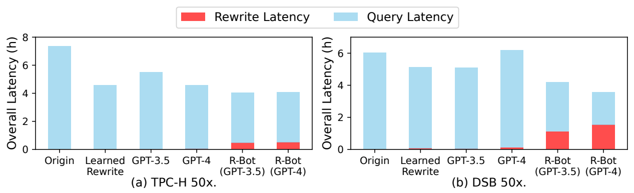


Figure 6: Comparison of Overall Latency.

TPC-H 10x (~10G) and TPC-H 50x (~50G). (*ii*) DSB is a more complex OLAP benchmark adapted from TPC-DS [18], which contains 429 columns and 76 queries. We also test *R-Bot* on datasets DSB 10x (~10G) and DSB 50x (~50G). (*iii*) Calcite is constructed from Apache Calcite's rewrite rule test suites [2], which is widely used in evaluating query rewrite capabilities [14, 20, 49]. In our experiment, we select 44 representative queries with great rewrite potentials on a schema of 43 columns. [2] Additionally, we evaluate *R-Bot* with a 10G data size across distinct data distributions, namely Calcite (uni) for uniform distribution and Calcite (zipf) for Zipfian distribution.

Rewrite Evidences. Our prepared rewrite evidences are shown in Figure 5, including 67 expert-verified rewrite rule specifications (30 from 80k+ tokens of documents, 37 from 30k+ lines of code), and 2091 filtered rewrite Q&As from millions of website Q&As. Together, they cover diverse SQL operations (e.g., joins, sub-queries) and support a wide range of rewrite scenarios.

LLMs. We use the state-of-the-art LLM gpt-4o, and alternatively cheaper gpt-3.5-turbo-0125 [8]. We use the default temperature of 0.1, in balance of reproduction and LLM performance [7].

Evaluation Metrics. We evaluate *R-Bot* using two metrics. (*i*) Query Latency, which measures the execution time of the rewritten query. (*ii*) Overall Latency, which captures the total time spent on both rewriting and executing the query. For each query, we conduct five executions and calculate the average, excluding the highest and lowest values. For each metric, we evaluate performance based on the average, median, and 90th percentile (p90).

Evaluated Methods. The evaluated methods include: (*i*) *R-Bot (GPT-4)* is *R-Bot* using gpt-4o. (*ii*) *R-Bot (GPT-3.5)* is *R-Bot* using gpt-3.5-turbo-0125. (*iii*) *LearnedRewrite* employs Monte Carlo Tree Search (MCTS) to explore the optimal sequence of rewrite rules [61]. It estimates the performance improvements of the rewritten queries with query cost models, which guides the search process. [3] (*iv*) *GPT-4* without using techniques in *R-Bot*, which inputs the SQL query and the rewrite rules, and outputs the arranged rule sequence. (*v*) *GPT-3.5* similarly without using techniques in *R-Bot*.

7.2 Performance Comparison

We compare *R-Bot* with two types of baselines, including learning-based methods (*LearnedRewrite*), and origin LLMs (*GPT-4*, *GPT-3.5*).

Query Latency Reduction. Table 4 shows the query latency of rewritten queries. *R-Bot* outperforms the other query rewrite methods across all the datasets and metrics. The reasons are two-fold.

First, *R-Bot* can judiciously retrieve relevant rule specifications and Q&As to figure out the rewrite rules pertinent to the input query. For instance, *R-Bot* can apply the exploration rule "AGGREGATE_EXPAND_DISTINCT_AGGREGATES_TO_JOIN" weakly related to the input query, using retrieved evidences to achieve a 5.6x optimization. However, the other methods fail to identify this critical rule without guidance from rewrite evidences. Besides, we find that *GPT-4* and *GPT-3.5* tend to select only a small number of rules (e.g., 1), which is often sub-optimal. That is because, they exhibit hallucination without rewrite evidences, and thus miss the intricate correlations between rewrite rules and the input query.

Second, *R-Bot* adopts a step-by-step LLM rewrite method, which can leverage rewrite evidences to understand the interrelations among the rewrite rules during rule arrangement. For instance, *R-Bot* can accurately select and arrange multiple rewrite rules (e.g., 6) from analysis of retrieved evidences, so that they can co-operate together to rewrite the input query to an execution-efficient form (e.g., 4.6x accelerated). On the contrary, since *LearnedRewrite* relies on blind exhaustive search, it can hardly find the optimal rule sequence among tremendous possibilities (e.g., only 4.3x accelerated).

Overall Latency Reduction. We also evaluate overall latency on larger TPC-H 50x and DSB 50x datasets. As shown in Figure 6, *R-Bot* achieves the best latency, demonstrating improvements of 1.82x and 1.68x respectively. The reasons are three-fold. First, *R-Bot* can still optimize the query latency by finding better plans, thus outperforming other methods. Second, *R-Bot* takes some time (e.g., average around 1 min) to retrieve beneficial evidences and iteratively instruct LLM to select and arrange rewrite rules. Thus it can steadily identify the critical rewrite rules (e.g., "FILTER_SUB_QUERY_TO_JOIN") with much higher query latency reduction. Instead, other methods do not spend efforts understanding rewrite evidences, which often causes sub-optimal rewritten queries. Third, compared with results of TPC-H 10x and DSB 10x (see Table 4), we find that *R-Bot* remains similar rewrite latency but demonstrates higher query latency reduction proportional to data scale, which both results in superior overall latency.

Query Improvement Ratio. Table 5 shows the query-level latency reduction ratio by different query rewrite methods, with *R-Bot* ranging from about 21.0% to 88.6%. We find that *R-Bot* still outperforms

[2] For efficiency, we chose Calcite test queries with top 11% performance gains under Calcite rewrite. Additional experiments on full queries confirmed that *R-Bot* consistently reduces query latency and outperforms baseline methods.

[3] Learning-based query rewrite methods, including LLM-based approaches that require model training [36], are generally unsuitable for our tests due to the absence of a train-test dataset split. As for *LearnedRewrite*, we use the authoritative version [6], integrating query cost models approximated using query plan statistics.

Table 5: Comparison of Query Improvement Ratio.

Improve (#Queries)	TPC-H 10x	DSB 10x	Calcite (uni)
LearnedRewrite	7/44 (15.9%)	4/76 (5.3%)	29/44 (63.6%)
GPT-3.5	3/44 (6.8%)	4/76 (5.3%)	16/44 (36.4%)
GPT-4	6/44 (13.6%)	4/76 (5.3%)	21/44 (47.7%)
R-Bot (GPT-3.5)	**21/44 (47.7%)**	16/76 (21.0%)	31/44 (70.4%)
R-Bot (GPT-4)	17/44 (38.6%)	**18/76 (23.7%)**	**39/44 (88.6%)**

Table 6: Comparison of Query Latency on Calcite (zipf).

Query Latency (s)	Average	Median	p90
Origin	106.31	37.91	300.00
LearnedRewrite	71.24 (↓33.0%)	5.04 (↓86.7%)	300.00 (↓0.0%)
GPT-3.5	58.33 (↓45.1%)	20.04 (↓47.1%)	300.00 (↓0.0%)
GPT-4	61.80 (↑41.9%)	14.15 (↓62.7%)	300.00 (↓0.0%)
R-Bot (GPT-3.5)	32.44 (↓69.5%)	6.58 (↓82.6%)	57.40 (↓80.9%)
R-Bot (GPT-4)	**7.56 (↓92.9%)**	**4.96 (↓86.9%)**	**18.08 (↓94.0%)**

the other methods on the three datasets. That is because *R-Bot* can discover targeted beneficial rule arrangements using relevant evidences, while the other methods have difficulty specifying the optimal arrangement among tremendous search space.

Zero-Shot Robustness. First, as shown in Table 4, *R-Bot* outperforms the other query write methods on different datasets without any re-training. This is because the generalizability of pre-trained LLM enables *R-Bot* to automatically adapt to unseen database schema and workload. Besides, Table 6 further demonstrates the robustness of query rewrite methods on data distribution, which transfers the Calcite dataset from uniform distribution to zipf distribution. Compared with Calcite (uni) (see Table 4), we find that the performance of *R-Bot* remains consistent. *R-Bot* still achieves the lowest average, median, and p90 of query latency among the methods. That is because the rewrite evidences can cover different query rewrite scenarios, and *R-Bot* can adaptively decide among potential rule arrangements with feedbacks of database query cost.

Evaluation Across Various LLMs. As shown in Table 4, on most metrics, *R-Bot (GPT-3.5)* performs worse than *R-Bot (GPT-4)*, but still outperforms the other query rewrite methods. That is because *R-Bot* decomposes complex tasks into simpler stages, both in evidence retrieval and step-by-step LLM rewrite, where each stage can be still manageable even with less advanced LLMs like GPT-3.5.

7.3 Ablation Study

7.3.1 Rewrite Evidence Preparation. We evaluate the essence of rewrite evidence by comparing *GPT-4* with naive RAG. As shown in Table 7, naive RAG equipped with evidences can outperform *GPT-4* across all the metrics, because naive RAG can retrieve rewrite evidences that are somewhat relevant to the input query, which provides beneficial guidance in rewrite rule selection and ordering.

7.3.2 Structure-Semantics Evidence Retrieval. To evaluate the effectiveness of hybrid structure-semantics Q&A retrieval, we evaluate *R-Bot* separately with semantics-only and structure-only retrieval. As indicated in Table 8 , both methods exhibit a decline in performance compared to *R-Bot*. This reduction is due to their inability to identify Q&As beneficial for query rewrite from a holistic perspective, as they either overlook query semantics or structure information. For example, compared with *R-Bot*, they may miss key evidence, leading to the omission of critical rules like "AGGREGATE_EXPAND_DISTINCT_AGGREGATES" in arrangement.

Table 7: Ablation Study of Rewrite Evidence on Calcite (uni).

Query Latency (s)	Average	Median	p90
Naive RAG	**39.13**	**14.88**	**98.15**
GPT-4	60.86	20.06	300.00

Table 8: Ablation Study of Evidence Retrieval on Calcite (uni).

Query Latency (s)	Average	Median	p90
Structure-Semantics Retrieval	**12.45**	**5.04**	**48.30**
Structure-Only Retrieval	31.96	5.30	56.40
Semantics-Only Retrieval	39.45	8.37	65.67
Naive RAG	39.13	14.88	98.15

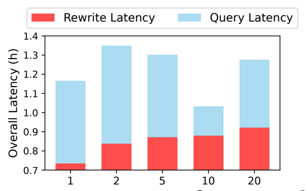


Figure 7: Impact of Retrieval Top-k on Calcite (uni).

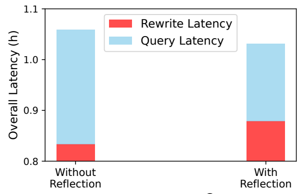


Figure 8: Impact of Rewrite Reflection on Calcite (uni).

Table 9: Ablation Study of LLM Rewrite on Calcite (uni).

Query Latency (s)	Average	Median	p90
Step-by-Step LLM Rewrite	**12.45**	**5.04**	**48.30**
One-Step LLM Rewrite	36.47	13.83	69.43

Retrieval Top-k. We also evaluate the impact of retrieval top-k, which decides the number of retrieved Q&As. As shown in Figure 7, we have two observations. First, rewrite latency grows with k, because *R-Bot* tends to generate more recipes with a greater number of retrieved Q&As. Consequently, as the recipes compose longer LLM contexts, each invocation of LLM requires additional time to process the extended context. Second, query latency decreases when k increases from 1 to 10, but rises again beyond 10. The reasons are two-fold. (*i*) When k is small, the additional retrieved Q&As are more likely to cover essential evidence, allowing *R-Bot* to select critical rules (e.g., "AGGREGATE_JOIN_TRANSPOSE"). (*ii*) For large k, it is likely to already cover the essential evidence in the retrieved Q&As, making further increase unnecessary as it may not provide additional information. Besides, increasing k too much may introduce noisy contexts, which may impair performance of LLM in query rewrite. Therefore, we set $k = 10$ for the other experiments. Meanwhile, we achieve a Precision@10 of 19.8%, indicating that on average at least one relevant Q&A is successfully retrieved.

7.3.3 Step-by-Step LLM Rewrite. We compare our step-by-step LLM rewrite with one-step LLM rewrite that instructs LLM to arrange the rules in one step using retrieved evidences, and evaluate the performance. As shown in Table 9, we find that one-step LLM rewrite often selects a smaller number of rules, which results in sub-optimal rule arrangements. That is because, given the extremely long context of retrieved evidences and available rewrite rules, LLM tends to produce short-cut solutions overlooking many critical details, and thus only selects the rules explicitly related to the input query. Instead, *R-Bot* employs a step-by-step LLM rewrite for reasoning, enabling the selection of higher-quality rules.

7.3.4 Rewrite Reflection. We also evaluate the impact of rewrite reflection in *R-Bot*. As shown in Figure 8, we have two observations. First, the rewrite latency increases due to the additional overhead introduced by rewrite reflection and LLM self-correction. Second,

Query081 of DSB 10x		
Query: WITH customer_total_return AS (SELECT ..., ca_state AS ctr_state, SUM(cr_return_amt_inc_tax) AS ctr_total_return FROM catalog_returns, …, customer_address WHERE ... GROUP BY ...) SELECT ... FROM customer_total_return ctr1, customer_address, … WHERE ctr1.ctr_total_return > (SELECT AVG(ctr_total_return)*1.2 FROM customer_total_return ctr2 WHERE ctr1.ctr_state = ctr2.ctr_state) AND ca_state = 'MI' AND ... ORDER BY ... LIMIT 100; **Latency:** 300.00 s		
Method	**Used Rewrite Rules**	**Query Latency (s)**
LearnedRewrite	FILTER_SUB_QUERY_TO_JOIN, AGGREGATE_PROJECT_MERGE, AGGREGATE_REDUCE_FUNCTIONS, AGGREGATE_JOIN_TRANSPOSE, …	19.46 (↓93.5%)
GPT-4	None	300.00 (↓0.0%)
R-Bot (GPT-4)	FILTER_INTO_JOIN, FILTER_SUB_QUERY_TO_JOIN, AGGREGATE_PROJECT_MERGE	**2.83 (↓99.0%)**

TestJoinConditionPush6 of Calcite (uni)		
Query: SELECT * FROM emp e RIGHT JOIN dept d ON e.deptno = d.deptno AND e.empno = d.deptno; **Latency:** 15.50 s		
Method	**Used Rewrite Rules**	**Query Latency (s)**
LearnedRewrite	PROJECT_REMOVE	15.50 (↓0.0%)
GPT-4	None	15.50 (↓0.0%)
R-Bot (GPT-4)	JOIN_CONDITION_PUSH	**1.66 (↓89.3%)**

Figure 9: Example Query Rewrite Results.

the query latency decreases with further rewrite reflection, leading to an overall latency reduction. That is because rewrite reflection helps mitigate the hallucination issue in LLMs by exploring multiple rewrite rule arrangements to find the most beneficial query rewrite.

7.4 Case Study

We provide some representative examples to illustrate the query rewrite results. As shown in Figure 9, *R-Bot* outperforms other methods by accurately identifying critical rules "FILTER_INTO_JOIN" and "FILTER_SUB_QUERY_TO_JOIN" in the first case study, which pushes down conditions involving only one join table and removes the time-consuming sub-queries in the query. On the contrary, *GPT-4* selects no useful rewrite rules due to absence of rewrite evidences and dedicated LLM algorithm design. Besides, *LearnedRewrite* tends to select an excessive number of rewrite rules under erroneous guidance, leading to a less efficient rewritten query. In the second case study, *R-Bot* also demonstrates superior performance by selecting the rewrite rule "JOIN_CONDITION_PUSH", which derives an additional condition "$e.deptno = e.empno$". This condition can be pushed down into the sub-query involving the table "*emp*", a refinement overlooked by other query rewrite methods. However other methods cannot detect this rule.

7.5 Deployment at Huawei with Real Customers

We have also deployed *R-Bot* at Huawei, which is empowered by open-sourced LLM DeepSeek-R1-Distill-Qwen-32B [4] and embedding model gte-Qwen2-1.5B-instruct [5]. We verify its effectiveness using a real-world dataset on the largest bank in China, containing 190 columns and 74GB data, as well as 20 real-world slow queries with 7 joined tables and 3 sub-queries on average. As shown in Figure 10, we find that *R-Bot* can effectively optimize these queries, i.e., improving the latency of 70% queries and reducing the overall latency from 9.23 hours to 4.37 hours. Notably, *R-Bot* remains effective even when (i) the tested queries are absent from the LLM pre-training corpus (not publicly available) and (ii) open-sourced 32B LLM and embedding model are used as the backend.

8 RELATED WORK

Query Rewrite. Existing query rewrite methods mainly adopt two paradigms. (1) Heuristic-based methods [9, 10, 15, 24, 25]. Some methods (e.g., PostgreSQL [10]) apply rewrite rules in a fixed order, which often overlook better orders in different scenarios. Besides, other heuristic-based methods (e.g., Volcano [25]) attempt to explore different rule orders with the aid of heuristic acceleration. However, by neglecting inter-dependencies among the rules, these methods perform a blind search, often failing to identify the optimal orders within reasonable time. (2) Learning-based methods [36, 61]. *LearnedRewrite* decides the rule order using Monte Carlo Tree Search guided by learned cost models. However, the learned model cannot be transferred to unseen database schema without additional retraining, which is often impractical. Hence, it calls for a query rewrite system capable of reliably identifying an appropriate sequence of rewrite rules to optimize query rewrite. A recent LLM-based query rewrite method addresses hallucination by incorporating query rewrite examples into the context for LLM to emulate [36]. However, it still lacks robustness, as it depends on an example pool generated from the training dataset and utilizes a trained model to select examples based on SQL query similarity.

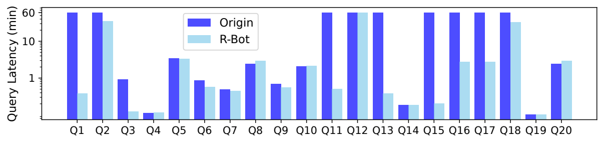


Figure 10: Performance of *R-Bot* on Real Dataset.

Automatic Rewrite Rule Discovery. Given the tedious effort required to curate a large number of high-quality rewrite rules, some methods have been recently proposed to automate the process of rule discovery [19, 20, 49]. *WeTune* [49] first represents the condition and transformation of rewrite rule with operators and symbols. Then it enumerates potential rewrite rules and verifies rule equivalence using SMT-solvers [16, 57]. *SlabCity* [20] discovers possible query rewrites for a specific query by recursively applying various transformations to the original query. Complementary to the above works, *R-Bot* retrieves highly relevant evidences to instruct LLM, effectively arranges newly discovered rules, and achieves superior adaptability and robustness in rule selection and ordering.

LLM for Database. There are also many studies to use LLMs to optimize databases [13, 22, 29, 37, 39, 44, 60, 62, 64]. For instance, *GPTuner* [29] enhances database knob tuning using LLM by leveraging domain knowledge to identify and tune knobs. *D-Bot* [44, 62] is an LLM-based database diagnosis system. Unlike these approaches, *R-Bot* is a novel query rewrite system powered by LLM.

9 CONCLUSION

We proposed an LLM-based query rewrite system. First, we prepared rewrite evidences from diverse sources, including rewrite rule specifications and rewrite Q&As. Next, we proposed a hybrid structure-semantics retrieval method to retrieve relevant rewrite evidences, based on which we generated rewrite recipes to instruct LLM for query rewrite. Then, we proposed a step-by-step LLM method, which iteratively utilized the retrieved Q&As and rewrite recipes to select and arrange rewrite rules with self-reflection. Experimental results demonstrated that *R-Bot* achieved remarkable improvements over existing methods. Moreover, *R-Bot* deployed at Huawei and with real customers showed the effectiveness of *R-Bot*.

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (62525202, 62232009), Shenzhen Project (CJGJZD20230724093403007), Zhongguancun Lab, Huawei, and Beijing National Research Center for Information Science and Technology (BNRist). Guoliang Li is the corresponding author.

REFERENCES

[1] 2018. *JavaSymbolSolver*. Retrieved October 17, 2024 from https://github.com/javaparser/javasymbolsolver

[2] 2024. *Apache Calcite*. Retrieved September 23, 2024 from https://github.com/apache/calcite

[3] 2024. *Chroma*. Retrieved October 17, 2024 from https://www.trychroma.com

[4] 2024. *DeepSeek-R1*. Retrieved December 24, 2024 from https://github.com/deepseek-ai/DeepSeek-R1

[5] 2024. *gte-Qwen2-1.5B-instruct*. Retrieved December 24, 2024 from https://huggingface.co/Alibaba-NLP/gte-Qwen2-1.5B-instruct

[6] 2024. *LearnedRewrite: An online logical query rewrite demo (schema+sql only)!* Retrieved December 24, 2024 from https://github.com/zhouxh19/LearnedRewrite/tree/main

[7] 2024. *LlamaIndex, Data Framework for LLM Applications*. Retrieved October 17, 2024 from https://www.llamaindex.ai/

[8] 2024. *Models - OpenAI API*. Retrieved October 17, 2024 from https://platform.openai.com/docs/models

[9] 2024. *MySQL*. Retrieved October 17, 2024 from https://www.mysql.com/

[10] 2024. *PostgreSQL: The world's most advanced open source database*. Retrieved October 17, 2024 from https://www.postgresql.org/

[11] 2024. *sentence-transformers/multi-qa-mpnet-base-cos-v1 · Hugging Face*. Retrieved October 17, 2024 from https://huggingface.co/sentence-transformers/multi-qa-mpnet-base-cos-v1

[12] 2024. *Stack Overflow - Where Developers Learn, Share, & Build Careers*. Retrieved October 17, 2024 from https://stackoverflow.com

[13] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. 2023. Gpt-4 technical report. *arXiv:2303.08774* (2023).

[14] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2911–2924.

[15] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*. 221–230.

[16] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *CIDR*. 1–7.

[17] Gordon V Cormack, Charles LA Clarke, and Stefan Buettcher. 2009. Reciprocal rank fusion outperforms condorcet and individual rank learning methods. In *SIGIR*. 758–759.

[18] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A decision support benchmark for workload-driven and traditional database systems. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3376–3388.

[19] 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. *SIGMOD* 1, 4 (2023), 1–26.

[20] 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.

[21] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2024. Evaluating large language models in class-level code generation. In *ICSE*. 1–13.

[22] Ju Fan, Zihui Gu, Songyue Zhang, Yuxin Zhang, Zui Chen, Lei Cao, Guoliang Li, Samuel Madden, Xiaoyong Du, and Nan Tang. 2024. Combining Small Language Models and Large Language Models for Zero-Shot NL2SQL. *VLDB* 17, 11 (2024), 2750–2763.

[23] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997* (2023).

[24] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[25] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*. IEEE, 209–218.

[26] Shengding Hu, Yuge Tu, Xu Han, Chaoqun He, Ganqu Cui, Xiang Long, Zhi Zheng, Yewei Fang, Yuxiang Huang, Weilin Zhao, et al. 2024. Minicpm: Unveiling the potential of small language models with scalable training strategies. *arXiv preprint arXiv:2404.06395* (2024).

[27] Lei Huang, Weijiang Yu, Weitao Ma, Weihong Zhong, Zhangyin Feng, Haotian Wang, Qianglong Chen, Weihua Peng, Xiaocheng Feng, Bing Qin, et al. 2023. A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions. *arXiv preprint arXiv:2311.05232* (2023).

[28] Shima Imani, Liang Du, and Harsh Shrivastava. 2023. Mathprompter: Mathematical reasoning using large language models. *arXiv:2303.05398* (2023).

[29] Jiale Lao, Yibo Wang, Yufei Li, Jianping Wang, Yunjia Zhang, Zhiyuan Cheng, Wanghu Chen, Mingjie Tang, and Jianguo Wang. 2024. GPTuner: A Manual-Reading Database Tuning System via GPT-Guided Bayesian Optimization. *Proceedings of the VLDB Endowment* 17, 8 (2024), 1939–1952.

[30] Boyan Li, Yuyu Luo, Chengliang Chai, Guoliang Li, and Nan Tang. 2024. The Dawn of Natural Language to SQL: Are We Fully Ready? *Proceedings of the VLDB Endowment* 17, 11 (2024), 3318–3331.

[31] Dacheng Li, Rulin Shao, Anze Xie, Ying Sheng, Lianmin Zheng, Joseph Gonzalez, Ion Stoica, Xuezhe Ma, and Hao Zhang. 2023. How Long Can Context Length of Open-Source LLMs truly Promise?. In *NeurIPS 2023 Workshop on Instruction Tuning and Instruction Following*.

[32] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *VLDB* 17, 12 (2024), 3786–3798.

[33] Guoliang Li, Jiayi Wang, Chenyang Zhang, and Jiannan Wang. 2025. Data+ AI: LLM4Data and Data4LLM. In *Companion of the 2025 International Conference on Management of Data*. 837–843.

[34] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3028–3042.

[35] Guoliang Li, Xuanhe Zhou, and Xinyang Zhao. 2024. LLM for Data Management. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4213–4216.

[36] 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! ery Eff iciency. *Proceedings of the VLDB Endowment* 18, 1 (2024), 53–65.

[37] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. *arXiv preprint arXiv:2403.09060* (2024).

[38] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2024. Lost in the Middle: How Language Models Use Long Contexts. *ACL* 11 (2024), 157–173.

[39] Xinyu Liu, Shuyu Shen, Boyan Li, Peixian Ma, Runzhi Jiang, Yuyu Luo, Yuxin Zhang, Ju Fan, Guoliang Li, and Nan Tang. 2024. A Survey of NL2SQL with Large Language Models: Where are we, and where are we going? *arXiv preprint arXiv:2408.05109* (2024).

[40] Leland McInnes, John Healy, and James Melville. 2018. Umap: Uniform manifold approximation and projection for dimension reduction. *arXiv preprint arXiv:1802.03426* (2018).

[41] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*. 3982–3992.

[42] Parth Sarthi, Salman Abdullah, Aditi Tuli, Shubh Khanna, Anna Goldie, and Christopher D Manning. 2024. RAPTOR: Recursive Abstractive Processing for Tree-Organized Retrieval. In *ICLR*.

[43] Zhaoyan Sun, Jiayi Wang, Xinyang Zhao, Jiachi Wang, and Guoliang Li. 2025. Data Agent: A Holistic Architecture for Orchestrating Data+ AI Ecosystems. *arXiv preprint arXiv:2507.01599* (2025).

[44] Zhaoyan Sun, Xuanhe Zhou, Jianming Wu, Wei Zhou, and Guoliang Li. 2025. D-Bot: An LLM-Powered DBA Copilot. In *Companion of the 2025 International Conference on Management of Data*. 235–238.

[45] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *SIGMOD*. 1493–1509.

[46] Xiu Tang, Sai Wu, Mingli Song, Shanshan Ying, Feifei Li, and Gang Chen. 2022. PreQR: pre-training representation for SQL understanding. In *SIGMOD*. 204–216.

[47] Jiayi Wang and Guoliang Li. 2025. Aop: Automated and interactive llm pipeline orchestration for answering complex queries. CIDR.

[48] Jiayi Wang, Guoliang Li, and Jianhua Feng. 2025. iDataLake: An LLM-Powered Analytics System on Data Lakes. *IEEE Data Eng. Bull.* 49, 1 (2025), 57–69.

[49] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. Wetune: Automatic discovery and verification of query rewrite rules. In *SIGMOD*. 94–107.

[50] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *NeuIPS* 35 (2022), 24824–24837.

[51] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2024. Tree of thoughts: Deliberate problem solving with large language models. *NeuIPS* 36 (2024).

[52] Jintao Zhang, Guoliang Li, and Jinyang Su. 2025. SAGE: A Framework of Precise Retrieval for RAG. In *ICDE*. IEEE Computer Society, 1388–1401.

[53] Xinyang Zhao, Xuanhe Zhou, and Guoliang Li. 2024. Chat2data: An interactive data analysis system with rag, vector databases and llms. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4481–4484.

[54] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. Queryformer: A tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.

[55] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2023. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *VLDB* 17, 4 (2023), 823–835.

[56] Yihang Zheng, Bo Li, Zhenghao Lin, Yi Luo, Xuanhe Zhou, Chen Lin, Jinsong Su, Guoliang Li, and Shifu Li. 2024. Revolutionizing Database Q&A with Large Language Models: Comprehensive Benchmark and Evaluation. *arXiv preprint arXiv:2409.04475* (2024).

[57] Qi Zhou, Joy Arulraj, Shamkant B Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A symbolic approach to proving query equivalence under bag semantics. In *ICDE*. IEEE, 2735–2748.

[58] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. Cracking SQL Barriers: An LLM-based Dialect Translation System. *SIGMOD* 3, 3 (2025), 1–26.

[59] Wei Zhou, Yuyang Gao, Xuanhe Zhou, and Guoliang Li. 2025. CrackSQL: A Hybrid SQL Dialect Translation System Powered by Large Language Models. *arXiv preprint arXiv:2504.00882* (2025).

[60] Xuanhe Zhou, Junxuan He, Wei Zhou, Haodong Chen, Zirui Tang, Haoyu Zhao, Xin Tong, Guoliang Li, Youmin Chen, Jun Zhou, et al. 2025. A Survey of LLM × DATA. *arXiv preprint arXiv:2505.18458* (2025).

[61] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *VLDB* 15, 1 (2021), 46–58.

[62] 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. *VLDB* 17, 10 (2024), 2514–2527.

[63] Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. 2023. A learned query rewrite system. *VLDB* 16, 12 (2023), 4110–4113.

[64] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. Db-gpt: Large language model meets database. *Data Science and Engineering* 9, 1 (2024), 102–111.