

# LLM-R<sup>2</sup>: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency

Zhaodonghui Li\*  
Nanyang Technological University,  
DAMO Academy Alibaba Group,  
Singapore  
G220002@e.ntu.edu.sg

Haitao Yuan  
Nanyang Technological University,  
Singapore  
haitao.yuan@ntu.edu.sg

Huiming Wang\*\*  
Singapore University of Technology  
and Design, Singapore  
huiming\_wang@mymail.sutd.edu.sg

Gao Cong  
Nanyang Technological University,  
Singapore  
gaocong@ntu.edu.sg

Lidong Bing  
DAMO Academy, Alibaba Group,  
Singapore  
l.bing@alibaba-inc.com

## ABSTRACT

Query rewrite, which aims to generate more efficient queries by altering a SQL query’s structure without changing the query result, has been an important research problem. In order to maintain equivalence between the rewritten query and the original one during rewriting, traditional query rewrite methods always rewrite the queries following certain rewrite rules. However, some problems still remain. Firstly, existing methods of finding the optimal choice or sequence of rewrite rules are still limited and the process always costs a lot of resources. Methods involving discovering new rewrite rules typically require complicated proofs of structural logic or extensive user interactions. Secondly, current query rewrite methods usually rely highly on DBMS cost estimators which are often not accurate. In this paper, we address these problems by proposing a novel method of query rewrite named LLM-R<sup>2</sup>, adopting a large language model (LLM) to propose possible rewrite rules for a database rewrite system. To further improve the inference ability of LLM in recommending rewrite rules, we train a contrastive model by curriculum to learn query representations and select effective query demonstrations for the LLM. Experimental results have shown that our method can significantly improve the query execution efficiency and outperform the baseline methods. In addition, our method enjoys high robustness across different datasets.

## PVLDB Reference Format:

Zhaodonghui Li\*, Haitao Yuan, Huiming Wang\*\*, Gao Cong, and Lidong Bing. LLM-R<sup>2</sup>: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. PVLDB, 14(1): XXX-XXX, 2020. doi:XX.XX/XXX.XX

## PVLDB Artifact Availability:

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.  
doi:XX.XX/XXX.XX

\*: Zhaodonghui Li is under the Joint PhD Program between DAMO Academy and Nanyang Technological University.

\*\* : Work done when interned in DAMO Academy.

The source code, data, and/or other artifacts have been made available at <https://github.com/DAMO-NLP-SG/LLM-R2>.

## 1 INTRODUCTION

With the rapid growth of data in various fields, it is common to take seconds or minutes, and even longer to execute an SQL query. Therefore, efficient query processing has been a crucial task in modern database systems. One of the key topics in query optimization that has gained significant attention is query rewrite [19, 23]. The objective of query rewrite is to output a new query equivalent to the original SQL query, while having a shorter execution time. Ideally, query rewrite should fulfill three critical criteria: (1) **Executability**: the rewritten query should be able to be executed without any errors; (2) **Equivalence**: it must yield identical results as the original query; (3) **Efficiency**: this encompasses two aspects—*Execution Efficiency* and *Computational Efficiency*. *Execution Efficiency* refers to the requirement that the rewritten query executes more efficiently than the original, while *Computational Efficiency* implies that the overhead of the rewriting process should be justifiable by the time savings achieved during query execution.

To enhance both **Executability** and **Equivalence** in rewritten queries, existing studies have predominantly concentrated on rule-based rewriting techniques. In particular, these studies are divided into two orthogonal research directions: the discovery of novel rewriting rules and the effective application of existing rules. For the first direction, although some studies have discovered more rewrite rules [6, 27, 29], there are many challenges related to the complexity of rule validation and the specificity of their applicability, often resulting in high computational demands and professional-level user competence. For example, Wetune [27] only supports discovering rewrite rules on limited types of operators and Query-booster [6] necessitates user engagement with specialized rule syntax, respectively. Therefore, this paper shifts focus toward the latter direction, delving into the methodologies for the effective utilization of pre-established rules. For example, Learned Rewrite [35] utilizes existing rewrite rules from the Apache Calcite [7] platform and learns to select rules to apply. It notably incorporates a Monte Carlo search algorithm in collaboration with a machine-learned query cost estimator to streamline the selection process. However, it’s non-trivial to solve the challenges related to the computational

demand of the Monte Carlo algorithm and the precision of the cost estimation model, which can significantly impact the **execution efficiency**.

On the other hand, with the rise of large language models (LLMs), there also exist some “large language model for database” projects [3, 30] that support direct query rewrite. The idea of these methods is to utilize the sequence-to-sequence generation ability of a language model to directly output a new rewritten query given an input query, without considering any rewrite rules or DBMS information. Although it is possible for these methods to discover new rewrites not following any existing rules, they easily suffer from the hallucination problem of language models [17, 32], especially for long and complicated queries, where language models give plausible but incorrect outputs. Either a syntax or reference error during generation will lead to vital errors when executing the query. Therefore, relying solely on LLM’s output query may violate the **executability** and **equivalence** to the original query, deviating from the basic aim for query rewrite.

To overcome the limits of the current query rewriting techniques and benefit from their advantages, we propose an LLM-enhanced rewrite system to use LLMs to suggest rewrite rule strategies and apply these strategies with an existing database platform to rewrite an input query. Inspired by the LLM-based learning framework for using tools [25, 31], we leverage the LLM’s strong generalization and reasoning abilities for query rewriting while avoiding issues like hallucination. We design a novel LLM-enhanced query rewrite system to automate the process of selecting more effective rewrite rules, note that the **executability** and **equivalence** of the rewritten query are guaranteed since all the candidate rules are provided by existing DB-based rule rewrite platforms. In addition to meeting the basic requirements of valid query rewrite, we also develop new techniques to boost the **executing efficiency** of our rewrite system. Firstly, to overcome hallucination, we collect a pool of demonstrations consisting of effective query rewrites using existing methods and our designed baselines. We then learn a contrastive query representation model to select the most useful in-context demonstration for the given query to prompt the system, optimizing the LLM’s rewrite rule selection. In addition, to address the challenge of limited training data, we propose to utilize the learning curriculum technique [8] to schedule the training data from easy to hard. We apply our LLM-enhanced rewrite method on three different datasets, namely TPC-H, IMDB, and DSB. We observe a significant query execution time decrease using our method, taking only 52.5%, 56.0%, 39.8% of the querying time of the original query and 94.5%, 63.1%, 40.7% of the time of the state-of-the-art baseline method on average on the three datasets.

Our main contributions are:

- To the best of our knowledge, we are the first to propose an LLM-enhanced query rewrite system that can automatically select effective rules from a given set of rewrite rules to rewrite an input SQL query.
- To enable LLMs to select better rewrite rules for a query, we construct a demonstration pool that contains high-quality demonstrations so that we can select good demonstrations to prompt the LLM-enhanced rewrite system for few-shots learning.

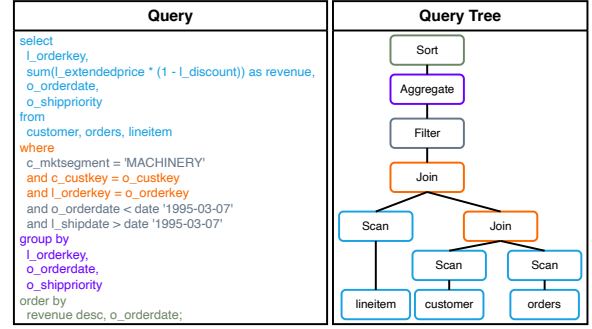


Figure 1: A TPC-H query and its query tree

- We learn a contrastive query representation model to optimize the demonstration selection. To overcome the challenge of limited training data, we further design a learning curriculum to schedule the training data from easy to hard.
- We further analyze the robustness of our method. By applying our method to unseen datasets and different dataset volumes, we demonstrate that our method is much more flexible than the baseline methods and shed light on generalizing to other database problems.

## 2 PRELIMINARY

In this section, we first introduce some key concepts including query, query tree and query rewrite rules in Section 2.1. Then, we will formalize the problem of query rewrite based on rules in Section 2.2. Finally in Section 2.3, we introduce the related work.

### 2.1 Query and Rewrite Rules

**Query & Query tree.** Each query in our study is formulated as an executable SQL statement. Furthermore, we model each query as a query tree using various nodes, where each node represents a specific type of query operator (e.g., Sort, Join, and Scan). Figure 1 illustrates an example of a SQL query and its corresponding query tree representation. It is worth noting that any given query can be transformed into a query tree, and conversely, this query tree can be reverted back to its original raw query form.

**Query rewrite rules.** Given an input query denoted as  $Q$ , a sequence of transformation methods, represented as  $r_1, r_2, \dots$ , can be applied to the query’s query tree, yielding an equivalent query, denoted as  $Q^*$ . These transformation methods, referred to as rewrite rules, encompass a diverse range of functionalities. These include the conversion of one operator to another, the alteration of execution sequences between operators, and the elimination of redundant operators. Table 1 delineates a representative set of these query rewrite rules. For the sake of brevity, we succinctly express the query rewrite process as  $Q^* = R(Q)$ , where  $R = [r_1, r_2, \dots, r_n]$  symbolizes the sequence of  $n$  applied rewrite rules.

### 2.2 Rule-based Query Rewrite

With the introduction of the rewrite rules, we now formally define the problem of query rewrite based on rules as follows:

*Definition 2.1.* (Rule-based query rewrite): Consider an input query  $Q$  and a set of candidate rewrite rules  $R$ . The objective is to identify a sequence of rules  $R^* = [r_1^*, r_2^*, \dots, r_n^*]$  where  $r_i^* \in R$ ,

**Table 1: Examples of query rewrite rules. Examples of query rewrite rules of the Apache Calcite Rules [1].**

Rule Name	Rule Description
AGGREGATE_UNION_AGGREGATE	Rule that matches an Aggregate whose input is a Union one of whose inputs is an Aggregate
FILTER_INTO_JOIN	Rule that tries to push filter expressions into a join condition and into the inputs of the join
JOIN_EXTRACT_FILTER	Rule to convert an inner join to a filter on top of a cartesian inner join
SORT_UNION_TRANSPOSE	Rule that pushes a Sort past a Union

that transforms the query  $Q$  into a more efficient version  $Q^* = R^*(Q)$ . The efficiency of the rewritten query  $Q^*$  is quantified by its execution latency. Such rewrite is characterized by transforming  $Q$  into an equivalent query  $Q^*$ , which exhibits a lower execution latency compared to other possible rewritten versions of the query. The problem can be formally represented as:

$$\begin{aligned} \operatorname{argmin}_{R^* \subseteq R} \operatorname{latency}(Q^*) \\ \text{s.t. } Q^* = R^*(Q) \end{aligned} \quad (1)$$

## 2.3 Related Work

**2.3.1 Query Rewrite.** Query rewrite is a significant function in current Database Management Systems (DBMSs), and can be supported in the query optimizers [14–16]. In particular, DBMSs, such as Calcite [7] and PostgreSQL [4], have developed different rewrite functions to achieve various rewrite rules. Consequently, there are two primary research directions for the query rewriting problem: discovering new rewrite rules and optimally leveraging existing rewrite rules.

**Discovering New Rewrite Rules.** Recent advancements, exemplified by Querybooster [6] and Wetune [27], have made significant strides in discovering new rewrite rules through the application of relational algebra proofs [29]. Querybooster enables database users to suggest rules through a specialized rule language, facilitating the back-end generation and application of these rules for more adaptable rewriting. On the other hand, Wetune compiles potential rewrite templates and pinpoints constraints that convert these templates into actionable rules. While these methodologies have proven their worth by efficiently handling small real-world workloads, they have their limitations. Querybooster’s effectiveness hinges on the user’s ability to propose potent rules, whereas Wetune’s efficacy on simple or generalized queries remains uncertain.

**Selecting Rewrite Rules.** The heuristic rewrite approach executes rewrite rules contingent upon the types of operators involved. Nonetheless, this technique is not without flaws. It might not identify the most optimal sequences for rewriting and often lacks the mechanisms necessary for evaluating the benefits of such rewrites. To address this issue, Learned Rewrite [35] employs a Monte Carlo Tree search to optimize the selection of applicable rules. It conceptualizes each query as a query tree, with applied rules modifying the tree’s structure. This approach utilizes a learned cost model to predict the impact of applying specific rules, enabling the selection of an optimal rewrite sequence through Monte Carlo Tree search. While this method improves adaptability to varying queries and database structures, it faces challenges in cost model accuracy and potential local minima in the search process, highlighting areas for future enhancement in rule-based query rewriting techniques.

**2.3.2 LLM-based SQL Solvers.** Large Language Models (LLMs) have recently emerged as a hot topic in machine learning research, captivating the interest of many in the field due to their impressive capabilities. These models have demonstrated a surprisingly strong ability to handle a variety of text-related tasks, excelling in areas such as generation, decision-making, and deduction. One such task that is highly related to DB research is text-to-SQL, in which an LLM directly generates a SQL query given database information and user requirements. Numerous studies [20, 26, 36] have highlighted the potential of LLMs in the text-to-SQL task, showcasing their proficiency in SQL query-related tasks. While much of this existing research has focused on LLMs’ ability to generate executable queries, there is a growing recognition of the importance of other factors, particularly the efficiency and accuracy of these queries when applied in real-world scenarios. In particular, [20] discussed their attempts in an efficiency-oriented query rewrite task, where an LLM is directly given an input query and tries to rewrite it into a more efficient one.

However, a significant issue previous LLM-based face is the problem of hallucination, which refers to instances where the model generates output that is not only incorrect but is done so with a misleading level of confidence. This is particularly problematic in the context of database applications, where accuracy is paramount. Therefore, we propose a different direction of utilising the LLMs while overcoming hallucination. Instead of using LLM to directly output an SQL query, we adopted a DB-based SQL rewriter enhanced by an LLM.

**2.3.3 In-context Learning.** Due to the extensive data and resource requirements of fine-tuning an LLM, many works choose to utilize LLMs by the technique called in-context learning (ICL), where no modifications to the LLMs’ model weights are made. The concept of ICL, first introduced by Brown et al. in their seminal work on GPT-3 [9], shows that language models like GPT-3 can leverage in-context demonstrations at inference time to perform specific tasks, without updating the model weights. ICL typically involves enriching the context with select examples to steer the model’s output. Formally, consider a model denoted as  $M$  and a contextual input represented by  $P$ . The output  $o$  generated by applying the ICL method to model  $M$  with input  $P$  can be succinctly expressed as  $o = ICL_M(P)$ .

ICL has rapidly gained popularity for addressing diverse challenges in natural language processing. However, it is a sophisticated technique requiring careful implementation. Extensive research, including studies by [28] and [21], has explored the intricacies of LLMs’ learning processes in this context. These studies highlight that the success of in-context learning is closely related to the construction of the context and the quality of the examples used.

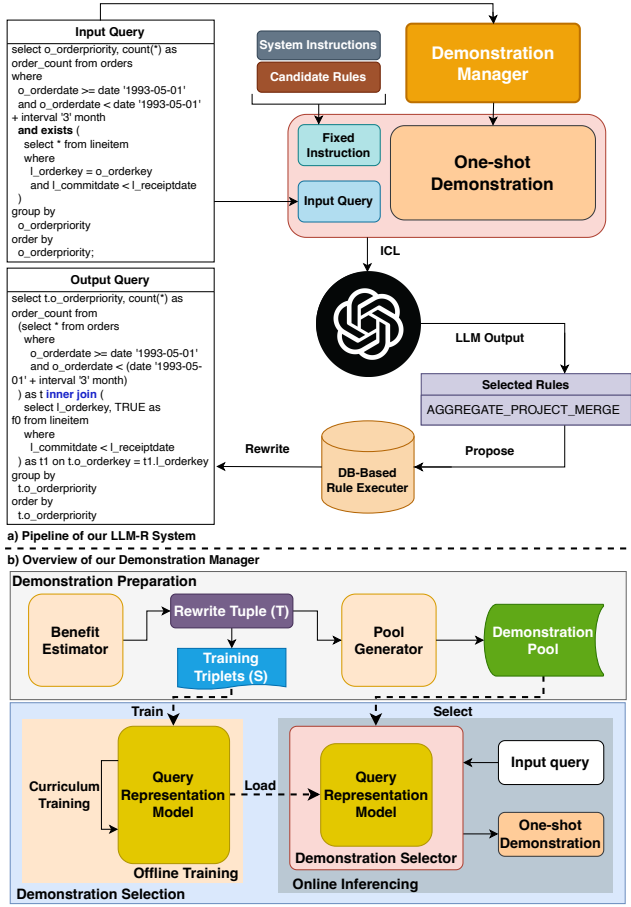


Figure 2: The Framework of LLM-enhanced Rewrite System

### 3 LLM-ENHANCED REWRITE SYSTEM

In this section, we will introduce our innovative LLM-enhanced rule-based rewrite system (LLM-R<sup>2</sup>). In Section 3.1, we will first illustrate the pipeline of our rewrite system. Then in Section 3.2, we will state our motivation to optimize the demonstration selection and introduce our novel *Demonstration Manager* module.

#### 3.1 System Pipeline

As shown in Figure 2(a), the system integrates an LLM into the query rewrite system utilizing the ICL methodology [9]. We construct the ICL prompt with three main components:

**Input query:** We employ the SQL statement corresponding to the provided input query  $Q$  for the prompt construction.

**Fixed instruction:** The fixed instruction consists of a system instruction  $I$  and a rule instruction  $R$ . While the system instruction specifies the task requirements, the rule instruction includes a comprehensive list of all candidate rewrite rules available for the language model to select. Each rule is accompanied by a concise explanation, enabling informed decision-making.

**One-shot demonstration:** Similar to directly using LLMs to rewrite queries, selecting rewrite rules using LLMs may also easily suffer from the hallucination problem, like outputting non-existing rules. To mitigate this and ensure the LLMs’ outputs are more closely

aligned with our task requirements, yielding superior rule suggestions, we use the demonstration as a part of the prompt. Formally, we define our demonstration given to the LLM-R<sup>2</sup> system as a pair of text  $D = \langle Q^D, R^D \rangle$ , where  $Q^D$  is the example query assembling the input query and  $R^D = [r_1^D, \dots]$  is the list of rules that have been applied to rewrite the example query. Such demonstrations can successfully instruct the LLM to follow the example and output a list of rewrite rules to apply on the new input query. In particular, this involves selecting a high-quality demonstration  $D$  from many successful rewritten demonstrations (i.e., denoted as a pool  $\mathcal{D}$ ) for each input query to guide the LLM effectively. To achieve this goal, we design a module named *Demonstration Manager*, whose details are elucidated in the subsequent section.

As specifically highlighted, Figure 3 delineates the prompt utilized within the In-Context Learning (ICL) process of our system. Upon constructing the prompt and feeding it into the LLM, we can extract a sequence of rewrite rules from the model’s output. These rules undergo further processing and execution by a database-based rule executor. For instance, the original input query in Figure 2(a) is modified by the “AGGREGATE\_PROJECT\_MERGE” rule, as highlighted in bold. This modification transforms the original query into a more optimized output query, demonstrating the practical application and effectiveness of the extracted rules in query optimization processes. Through the synergy of the LLM’s superior generalization capabilities and the rule executor’s precision, our proposed system guarantees extensive applicability, alongside ensuring the executability and equivalence of the rewritten queries. Consequently, this rewrite process can be formalized as follows:

**Definition 3.1.** (LLM-enhanced Query Rewrite): Given a large language model  $M$ , a textual instruction outlining the rewrite task  $I$ , a set of candidate rules  $R$ , one successful rewrite demonstration  $D$  selected from the demonstration pool  $\mathcal{D}$ , and an input query  $Q$ , a prompt  $P$  is constructed and provided as input to  $M$  as:

$$P = I \oplus R \oplus D \oplus Q$$

From  $M$ , a sequence of rewrite rules  $R^*$  is derived:

$$R^* = ICL_M(P)$$

By sequentially applying these rewrite rules  $R^*$ , we generate an optimally equivalent query, represented as  $Q^* = R^*(Q)$ .

#### 3.2 Demonstration Manager Overview

**Motivation.** In the above ICL process, optimizing the prompt  $P = I \oplus R \oplus D \oplus Q$  is crucial for improving the output quality of LLMs. Given the fixed settings of system instruction( $I$ ), rule instruction( $R$ ), and input query( $Q$ ), our optimization efforts focus primarily on the demonstration( $D$ ), which is chosen to enhance model performance. Recent studies on LLMs (e.g., [9, 28]) have underscored the positive impact of high-quality in-context demonstrations on LLM output, reducing the tendency of LLMs to produce hallucinatory content. As shown in Figure 4, our rewrite system exhibits similar effectiveness variability w.r.t. the demonstrations used, further emphasizing the necessity of optimizing demonstration selection for specific input queries. Therefore, it is an important problem to optimize the demonstration selected for a given input query. Particularly,

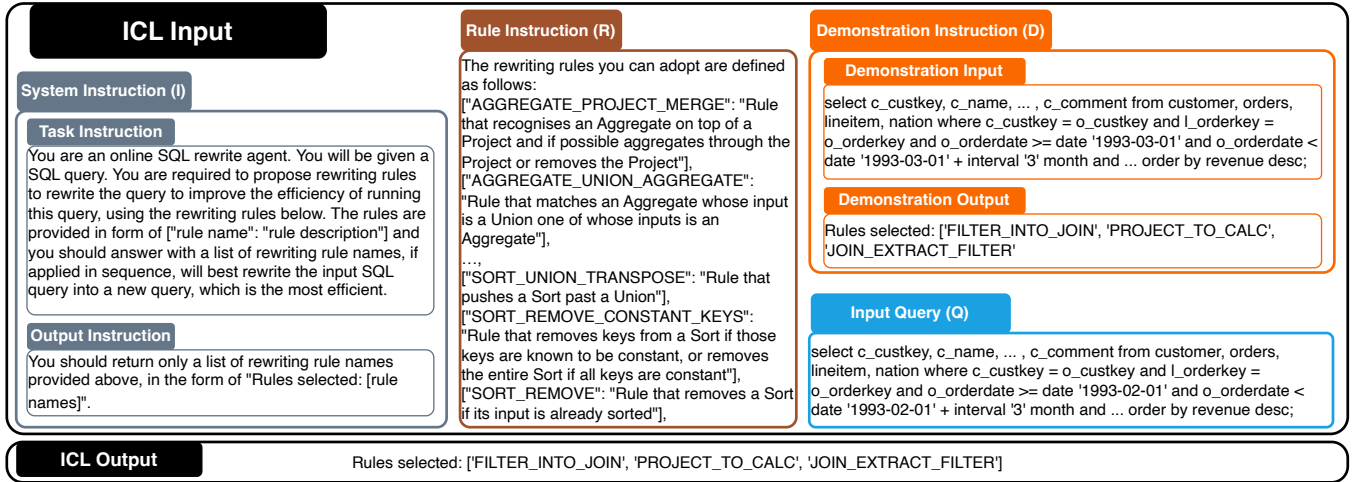


Figure 3: An Example of the In-Context Learning Process in LLM-R<sup>2</sup>. All the instructions are concatenated together as one string input to the LLM. In a zero-shot setting, the “Demonstration Instruction” will be removed and an input query will be appended directly after the “Rule Instruction”.

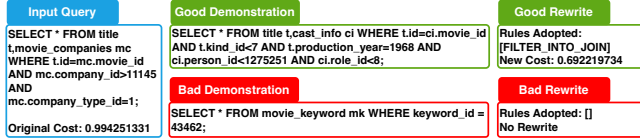


Figure 4: Example of good and bad demonstration selections

we address this problem by designing the *Demonstration Manager* module.

**Overview.** Figure 2(b) illustrates the basic structure of our proposed *Demonstration Manager* module, comprising two parts: *Demonstration Preparation* and *Demonstration Selection*.

(1) The primary objective of the *Demonstration Preparation* is to generate a substantial number of successful rewritten demonstrations for constructing a demonstration pool. Furthermore, this part also serves to supply training data essential for model learning in the second part. Specifically, we design two modules: the *Benefit Estimator* and the *Pool Generator*, to achieve our objectives. The *Benefit Estimator* is capable of assessing the potential benefits of a given query rewrite strategy, thereby generating corresponding rewrite tuple recording the performance of this rewrite strategy on the input query. Subsequently, the *Pool Generator* is employed to extract demonstrations for constructing a pool. Moreover, we utilize the rewrite tuples to derive training triplets, which are essential for model learning in subsequent parts.

(2) The second part involves the *Demonstration Selection* module, tasked with identifying the optimal demonstration from the pool for each input query. This process is enhanced by incorporating a query representation model within the selector, designed to evaluate the similarity between input queries and demonstrations in the pool. This representation model undergoes offline training using the training data. In addition, to obtain an effective model, we enhance the model’s training through the integration of a curriculum learning approach. Afterwards, the trained model is integrated into *Demonstration Selector* for online inference. In other words, upon receiving an input query for rewriting, the selector discerns and

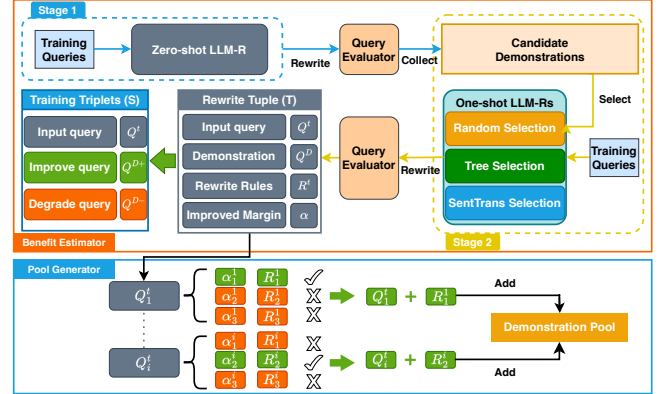


Figure 5: Our demonstration preparation module generates a set of training triplets and a demonstration pool.

selects the most appropriate demonstration from the pool based on the trained model. More detailed elaboration on the above two parts will be provided in the following sections.

## 4 DEMONSTRATION PREPARATION

In this section, we aim to generate sufficient high-quality data to build the demonstration pool. As shown in Figure 5, we first design the *Benefit Estimator* module to generate the ground truth, where each ground truth data point indicates the efficiency gain obtained by rewriting an input query using generated rules in the context of a demonstration. With sufficient ground truth, including both good and bad samples, we further design the *Pool Generator* module to select all good samples to build the demonstration pool. In addition, we can deduce contrastive training triplets from the ground truth, which can help train our selection model.

### 4.1 Benefit Estimator

Since we are only able to start with solely training queries without demonstrations, the triplet generation pipeline is segmented into



two distinct phases: the first stage involves initializing high quality candidate demonstrations utilizing baseline method and a zero-shot **LLM-R<sup>2</sup>** system where no demonstration is selected, followed by the demonstration adoption stage employing a one-shot **LLM-R<sup>2</sup>** system. Subsequently, each stage is elucidated in detail.

**Stage-1:** We start with a diverse set of input queries collected from our dataset as the training set. To obtain a rich set of effective rewrites as candidate demonstrations, we first apply our zero-shot LLM-enhanced rewrite system (LLM-R<sup>2</sup>) to rewrite the training set queries. After getting the rewrite rules adopted and the resulted rewrite queries, we directly execute the rewritten queries on the corresponding databases. The execution time of the rewritten queries as well as the original queries is evaluated to collect the initial candidate demonstration set consisting of the improvable queries, together with their rules adopted.

**Stage-2:** With the candidate demonstrations collected from the previous step, we can then estimate the benefits of these demonstrations when they are selected for a given input query. Motivated by [28], such improvable demonstrations are supposed to be more useful for the LLM to output improving rewrite suggestions, compared to using any degraded rewrite queries as demonstrations. In addition, the more “similar” the improving demonstration query is to the input query, the better output the LLM will generate. However, different from natural language inputs’ simple textual similarity, the similarity between SQL queries is indeed more complicated. To identify if the pool we collected truly contains high-quality and “similar” demonstrations for new input queries and refine the demonstration pool, we designed three heuristic demonstration-selection methods based on different levels of similarity as follows.

- **Random Selection:** A random demonstration query is selected from the candidate demonstrations for a given input query, where the similarity level lies on the same input category.
- **Tree Selection:** Query tree is an important structural feature for the queries, therefore, it is natural to align similarity with the query tree structure. We first compute the query trees of all the candidate demonstration queries, with operators as the tree nodes. Given an input query, we select the demonstration with the minimum tree edit distance from the input query tree within the candidate demonstrations.
- **SentTrans Selection:** At the textual level, we observe that queries are always considered as sentences for the language models to process. Based on the observation, we treat input queries as sentences and select the candidate demonstration query whose embedding is the most similar to the input query. Most of the effective LLMs are closed-sourced, which means we are not able to obtain the query embeddings of such LLMs. However, similar to LLMs, some small pre-trained language models share the same sequence-to-sequence mechanism, that the input text is first encoded by an encoder before feeding to the model. Using such encoders, like Sentence Transformers [24], we can obtain an embedding of a given sentence.

With the three demonstration selection methods above, we can prompt our **LLM-R<sup>2</sup>** system with the one-shot demonstration to obtain various rewrite results on the same training set. These new rewrite queries from the one-shot **LLM-R<sup>2</sup>** system are then evaluated in the same way as in Stage-1. Specifically, when we adopt one-shot demonstration to rewrite an input query  $Q^t$ , we are able to

estimate the benefit obtained from the demonstration by constructing the rewrite tuples (T) as  $\langle Q^t, D, R^t, \alpha \rangle$ , where  $Q^t$  represents a training query,  $D$  is the demonstration  $\langle Q^D, R^D \rangle$  selected for  $Q^t$ ,  $R^t$  denotes the adopted rules for  $Q^t$ , and  $\alpha$  represents the improved margin obtained by the query rewrite. In particular, given the original query cost  $C_0$  and the cost of rewritten query  $C_r$ , we define the improved margin as  $\alpha = C_0/C_r$ , where the larger margin the better rewrite result and larger benefit we have.

In addition, a set of training triplets is generated using the rewrite tuples obtained in preparation for training a contrastive representation model. For a given query  $Q^t$  in the rewrite tuple  $\langle Q^t, D, R^t, \alpha \rangle$ , we consider the demonstration query  $Q^D$  adopted as an improve query  $Q^{D+}$  for  $Q$ , if the improved margin  $\alpha > 1$ . In contrast, we denote the demonstration query as a degrade query  $Q^{D-}$  if  $\alpha < 1$ . If there are multiple improve(degrade) queries, we only select the one with the largest(smallest) improved margin. Since we have adopted multiple one-shot selection methods, now we are able to construct a training triplet for a given query as  $\langle Q^t, Q^{D+}, Q^{D-} \rangle$ . A set of training triplets can be further constructed if we enumerate the whole training query set.

## 4.2 Pool Generator

Apart from the training triplets, we also hope to prepare an effective demonstration pool so that our learned demonstration selection model can select demonstrations from it during online inference. The rewrite tuple generated by the *Benefit Estimator* module, recording the effectiveness of a sequence of rewrite rules  $R^t$  on an input query  $Q^t$ , naturally fits our need for a high-quality rewrite demonstration.

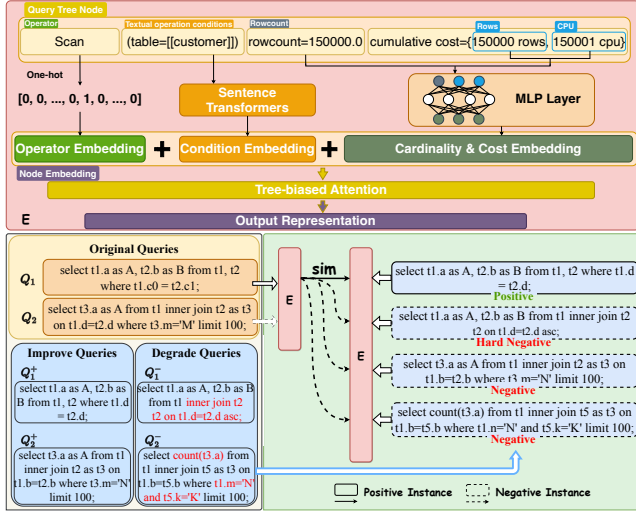
In particular, given the set of rewrite tuples generated by  $n$  input queries, we first separate them into  $n$  groups  $\{T_i\}_{1 \leq i \leq n}$  based on their corresponding input queries. Therefore, each group  $T_i$  can be represented as the tuple set  $T_i = \{ \langle Q_i^t, D_1^i, R_1^i, \alpha_1^i \rangle, \langle Q_i^t, D_2^i, R_2^i, \alpha_2^i \rangle, \dots \}$ . Since we have adopted various methods, multiple tuples have the same input query, and we only need the optimal rewrite rule sequence to form a demonstration for the query. Therefore, for each training query  $Q_i^t$  and its corresponding tuple group  $T_i$ , we only select the tuple with the largest improved margin, and the order is denoted as  $*$ , which can be formulated as follows:

$$\begin{aligned} * &= \underset{j \in [1, |T_i|]}{\operatorname{argmax}} \alpha_j^i \\ \text{s.t. } T_i &= \{ \langle Q_i^t, D_1^i, R_1^i, \alpha_1^i \rangle, \langle Q_i^t, D_2^i, R_2^i, \alpha_2^i \rangle, \dots \} \end{aligned} \quad (2)$$

Next, we construct the demonstration containing the input query and rules as the pair  $\langle Q_i^t, R_*^i \rangle$ , and then add the demonstration to the pool. As shown in Figure 5, when the largest improved margins  $\alpha_1^i$  and  $\alpha_2^i$  are identified for input queries  $Q_1^t$  and  $Q_2^t$ , the corresponding demonstrations  $\langle Q_1^t, R_1^1 \rangle$  and  $\langle Q_2^t, R_2^2 \rangle$  are selected with the rewrite rules  $R_1^1$  and  $R_2^2$  adopted.

## 5 DEMONSTRATION SELECTION

**Motivation.** Addressing the challenge of enhancing system performance, the selection of an optimal rewrite demonstration to guide the LLM for any given input query is required and remains uncertain. Intuitively, the greater the “similarity” between the input and demonstration queries, the more applicable the rewrite rule, thereby enhancing the LLM’s output efficacy. Therefore, to capture



**Figure 6: Our representation model and its contrastive training structure.** Each node of the query tree is encoded into a fixed length vector, while the final representation of the query is obtained by applying a tree-biased attention over the tree nodes. Such model  $E$  is then trained with contrastive query tuples generated.

such “similarity”, we design a contrastive model to learn the representations of queries in this *Demonstration Selection* module, where better demonstration queries are to have more similar representations to the input query. Consequently, the demonstration query that exhibits the highest resemblance to the input query is selected for the LLM, optimizing the generation of more effective outputs. **Overview.** In order to learn a contrastive representation model efficiently and effectively, the selection module consists of two main components: our contrastive model and a curriculum learning pipeline to improve the model training. We will first outline the representation model and its contrastive learning structure in Section 5.1, followed by a detailed discussion of the whole model learning pipeline in Section 5.2.

## 5.1 Contrastive Representation Model

As shown in Figure 6, our representation model  $E$  is constructed as a query encoder to encode the information describing a query, and a contrastive training structure to further train the encoder given training data. In particular, the information of a query tree is first encoded by nodes into node embeddings. A tree-biased attention layer will then compute the final representation of the query given the node embeddings. Such an encoder  $E$  is then trained using the contrastive learning structure drawn below it.

**Query encoder.** The representation of a query should focus on various key attributes, like the query tree structure and columns selected. Therefore, we design an encoder following [33] to take the query trees generated by DBMS’ query analyzer as inputs. It is notable that the original encoding in [33] utilizes the physical query plan which contains richer information, so that the objective of estimating query cost can be successfully achieved. Since we aim to capture the similarity between queries, we separately encode

the following information for each query tree node instead in our encoder, as shown in the top half of Figure 6:

- **Operator type:** We use one-hot encoding to encode the operator types into one vector, with value one for the current node operator type and zero for the rest positions.
- **Operator conditions:** Within each node, the details for the operator are explained in parentheses, including sort order for “Sort” operator, selected column for “Scan” operator etc. Different from the physical plans used in [33], such information has no unified form for encoding. We consider the conditions as text and encode using a pre-trained Sentence Transformers encoder [24]. Such an encoder can capture the textual differences between conditions effectively and have unified embedding dimensions to simplify further analysis.
- **Cardinality and cost:** From [34] we observe that the estimated cardinality and cost are important in describing a query. We collect the row count and estimated cumulative cost values and normalise them through an MLP layer.

We simply concatenate the three information vectors together to be the encoded embedding for a node in the given query tree. We use the same tree Transformer model in [33] to get the final representation of a query given its tree nodes’ embeddings. The final representation of the whole query will be computed by the tree-biased attention module.

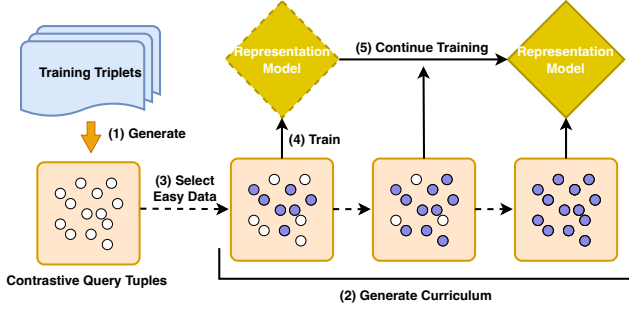
### Contrastive learning structure.

Due to the necessity of executing queries, the volume of training triplets produced by our demonstration preparation module is limited. Unlike the query representation model in [33], which is trained directly on abundant labeled data, our approach requires a more sophisticated training framework to effectively capture query representation with the generated training data. Inspired by SimCSE [13], we design a contrastive learning structure to train our query representation model on the limited training data. In a training batch containing  $N$  tuples, we consider each original query’s improved query as its “positive” query, its degraded query as its “hard negative” query, and the remaining improved and degraded queries within the same batch as “negative” queries. This allows us to pull close distances between original queries and their improved versions while pushing apart those with degraded queries. Following such setting, the loss  $l_i$  for the  $i_{th}$  tuple  $(Q_i, Q_i^+, Q_i^-)$  can be computed as

$$l_i = -\log \frac{e^{\text{sim}(h_i, h_i^+)/\tau}}{\sum_{j=1}^N (e^{\text{sim}(h_i, h_j^+)/\tau} + e^{\text{sim}(h_i, h_j^-)/\tau})} \quad (3)$$

where  $\tau$  is a temperature hyper-parameter,  $h_i$ ,  $h_i^+$  and  $h_i^-$  stand for the representation of  $Q_i$ ,  $Q_i^+$  and  $Q_i^-$  respectively, and the function  $\text{sim}(h_1, h_2)$  is the cosine similarity  $\frac{h_1^T h_2}{\|h_1\| \cdot \|h_2\|}$ .

As an example in a training batch of size 2, for the first original query  $Q_1$  shown in the bottom part of Figure 6, the positive query will be its corresponding improve query  $Q_1^+$ , and other in-batch improve or degrade queries  $Q_1^-$ ,  $Q_2^+$  and  $Q_2^-$  are all regarded as negative queries. The final loss for the batch will be the sum of the losses for the two tuples.



**Figure 7: The overall curriculum learning pipeline to train the contrastive selector using generated training triplets.**

## 5.2 Curriculum Learning Pipeline

**Motivation.** Although we have developed a representation-based demonstration selector, training the contrastive model presents several challenges. First, unlike the original SimCSE approach used in natural language inference tasks, which benefits from abundant data [10], our model’s training is constrained by data scarcity. Our contrastive query tuples, derived from a limited variety of training triplets, face scalability issues due to the high computational cost of query execution. Furthermore, the complexity of query representations in our model surpasses the simplicity of word embeddings used in SimCSE. Given these constraints—limited data and a complex training target—we propose adopting a curriculum learning pipeline. This approach is designed to enhance the learning efficiency and effectiveness of our contrastive representation model.

As depicted in Figure 7, the essence of this pipeline is to strategically implement an effective curriculum. Starting with the provided training triplets, we initially train our contrastive representation model on a smaller, simpler subset, progressively incorporating

easier subsets from the remaining dataset and retraining the model until all training data is utilized. The methodology for generating our curriculum is detailed in Algorithm 1. This algorithm begins with an empty model; each iteration involves selecting a subset of training data on which the current model performs with the highest confidence, followed by model retraining to incorporate this new subset (lines 5-17). This iterative retraining process continues until the entire training dataset has been incorporated.

In particular, we sample the easier subset of remaining training data by the confidence of the model to the data. Suppose we get the embeddings of two queries using our contrastive model to be  $x$  and  $y$ , we can compute their similarity scores using the cosine similarity to keep consistency with the training objective in Equation 3. For each contrastive query tuple  $\langle Q, Q^+, Q^- \rangle$ , since we expect to have the  $\text{sim}(E(Q), E(Q^+)) = 1$  and  $\text{sim}(E(Q), E(Q^-)) = 0$ , we define a confidence score of the contrastive model  $E$  to a given tuple as:

$$\text{conf}_E(Q) = \text{sim}(E(Q), E(Q^+)) - \text{sim}(E(Q), E(Q^-)) + 1 \quad (4)$$

Therefore, at each iteration  $i$ , given our trained model  $E_{i-1}$ , previous training dataset  $T_{i-1}$  and the unvisited dataset  $D_{i-1}$ , we can generate the current tuples (denoted as  $S_i$ ) with the highest confidence score in  $D_{i-1}$ . They are then moved into the training set, resulting in the new training set  $T_i = T_{i-1} + S_i$  and the new unvisited dataset  $D_i = D_{i-1} - S_i$ .

## 6 EXPERIMENT

In this section, we evaluate our proposed system’s effectiveness, efficiency, and generalization capabilities.

### 6.1 Experimental Setup

**6.1.1 Dataset.** We use three datasets from different domains for our evaluations:

**IMDB (JOB workload) [18]:** The IMDB [22] dataset consists of data on movies, TV shows, and actors. It’s utilized in conjunction with the Join Order Benchmark (JOB) to test a database management system’s efficiency in executing complex join queries, and it comprises 5,000 queries.

**TPC-H [5]:** A benchmark dataset for evaluating database management systems, generated using the official toolkit to include approximately 10 GB of data and 5,000 queries.

**Decision Support Benchmark (DSB) [11]:** This benchmark is developed to evaluate traditional database systems for modern decision support workloads. It is modified from the TPC-DS to include complex data distributions and challenging query templates, and it contains a total of 2,000 queries.

**6.1.2 Rewrite Rules.** To enhance the efficiency of the rule proposal and rewriting process for subsequent experiments, we integrate Apache Calcite [7] as our rewrite platform, alongside its comprehensive set of rewrite rules by following previous work [35]. Examples of utilized rewrite rules and their functions are illustrated in Table 1, with a complete enumeration available on the official website [1]. Specifically, we introduce a rule termed “EMPTY” to signify instances where the query remains unchanged, thereby standardizing LLM outputs with an indicator for scenarios that do not require query rewrite.

#### Algorithm 1 Contrastive Training under Curriculum Scheduler

**Require:** Total training data  $S_0$ , Number of iterations  $I$

**Require:** Initialized model  $E_0$

```

1:  $N_0 = \text{len}(S_0)$  ▷ Total number of training data
2:  $N = \lceil (N_0/I) \rceil$  ▷ Each iteration incremental data size
3:  $Tr_0 = \emptyset$  ▷ Initial training data
4:  $i = 1$  ▷ Initial iteration
5: while  $i \leq I$  do
6:   if  $\text{len}(N) > \text{len}(S_{i-1})$  then ▷ If less than N data left
7:      $Tr_i \leftarrow S_{i-1}$  ▷ Select all the data left
8:      $Tr_i = Tr_{i-1} + Tr_i$  ▷ Append to training data
9:     Train  $E_{i-1}$  on  $Tr_i$  and get  $E_i$  ▷ Continue train the model
10:  else
11:     $C_i \leftarrow \text{Top}_N(S_{i-1})$  based on  $\text{conf}_{E_{i-1}}(\cdot)$  ▷ Select N easy data following curriculum from the unvisited dataset
12:     $S_i = S_{i-1} - C_i$  ▷ Deduct them from unvisited data
13:     $Tr_i = Tr_{i-1} + C_i$  ▷ Append them to training data
14:    Train  $E_{i-1}$  on  $Tr_i$  and get  $E_i$  ▷ Train the model
15:     $i = i + 1$  ▷ Move to next iteration
16:  end if
17: end while
18: Use the final  $E_I$  for inference

```



Execution time(sec)	TPC-H				IMDB				DSB			
Method	Mean	Median	75th	95th	Mean	Median	75th	95th	Mean	Median	75th	95th
<b>Original</b>	70.90	22.00	37.01	300.00	6.99	1.86	5.12	32.49	60.55	6.64	26.55	300.00
<b>LR</b>	39.40	22.00	32.21	<b>159.95</b>	6.20	1.62	4.74	32.45	59.21	5.14	53.78	300.00
<b>LLM only</b>	70.67	22.00	37.01	300.00	6.96	1.86	5.10	32.49	61.60	6.53	26.40	300.00
<b>LLM-R<sup>2</sup> (Ours)</b>	<b>37.23</b>	<b>17.40</b>	<b>29.80</b>	164.12	<b>3.91</b>	<b>1.33</b>	<b>3.52</b>	<b>18.16</b>	<b>24.11</b>	<b>2.16</b>	<b>12.61</b>	<b>196.61</b>
% of <b>Original</b>	52.5%	79.1%	80.5%	54.7%	56.0%	71.3%	68.7%	55.9%	39.8%	32.5%	47.5%	65.5%
% of <b>LR</b>	94.5%	79.1%	92.5%	102.6%	63.1%	82.0%	74.3%	56.0%	40.7%	42.0%	23.4%	65.5%
% of <b>LLM only</b>	52.7%	79.1%	80.5%	54.7%	56.2%	71.3%	69.0%	55.9%	39.1%	33.1%	47.8%	65.5%

Table 2: Execution time v.s. different query rewrite methods

Counts	TPC-H/IMDB/DSB		
Method	Rewrite #	Improve #	Improve %
<b>LR</b>	258/203/456	192/197/193	74.42/97.04/42.32
<b>LLM only</b>	197/102/210	68/67/8	34.5/65.68/3.81
<b>LLM-R<sup>2</sup></b>	<b>323/302/341</b>	<b>305/292/222</b>	<b>94.43/96.69/65.10</b>

Table 3: The rewritten queries’ number v.s. different methods

**6.1.3 LLM Setting.** We leverage the ChatGPT API [2], which is built upon the GPT-3.5-turbo architecture [9]. Furthermore, we assess our system’s generalizability across other Large Language Models (e.g., GPT-4), as detailed in Section 6.5.

**6.1.4 Baseline Methods.** We compare our system with two baseline methods:

**Learned Rewrite (LR)** [35]: This approach, recognized as the state-of-the-art query rewrite method, incorporates a cost estimation model for predicting the performance of rewritten queries. It further employs a Monte Carlo Tree-based search algorithm to identify the optimal query.

**LLM only** [20]: This method straightforwardly generates a rewritten query from the input, incorporating task instructions, schema, and a fixed demonstration as prompts to the LLM. When the rewritten queries are not executable or equivalent to the original queries, we substitute them with the original queries. This ensures a fair comparison with rule-based methods.

**6.1.5 Training Setting.** In the demonstration preparation phase, we exclude any training queries already present in the demonstration pool from being selected as demonstrations to mitigate potential bias. For the development of our query representation-based demonstration selector, we adopt a curriculum learning strategy encompassing four iterations ( $I = 4$ ). Each iteration involves further training our contrastive representation model with a learning rate of  $10^{-5}$ , a batch size of 8, over three epochs, utilizing a Tesla-V100-16GB GPU.

**6.1.6 Evaluation Metrics.** For the evaluation of rewrite methods, two key metrics are employed: *query execution time* and *rewrite latency*, which are respectively employed to evaluate the executing efficiency and the computational efficiency. To mitigate variability, each query is executed five times on a 16GB CPU device, with the average execution time calculated after excluding the highest and lowest values. To address the challenge posed by overly complex queries that exceed practical execution times, a maximum time limit of 300 seconds is imposed, with any query exceeding this duration

Total (Latency)	TPC-H	IMDB	DSB
<b>LR</b>	40.98(1.58)	7.24(1.04)	60.99(1.78)
<b>LLM only</b>	75.37(4.70)	7.58(1.38)	64.21(6.00)
<b>LLM-R<sup>2</sup></b>	<b>40.63(3.40)</b>	<b>6.81(2.90)</b>	<b>27.40(3.29)</b>

Table 4: The rewrite performance in total average rewrite query execution time including the rewrite latency

Execution time(sec)	Mean	Median	75th	95th
<b>Original</b>	6.99	1.86	5.12	32.49
<b>LLM-R<sup>2</sup></b>	<b>4.41</b>	<b>1.35</b>	<b>3.57</b>	<b>17.84</b>
<b>LR</b>	-	-	-	-
<b>LLM only</b>	6.99	1.86	5.12	32.49

Table 5: Training on TPC-H and Testing on IMDB

assigned a default execution time of 300 seconds. This approach facilitates a broader range of experimental conditions. For assessing rewrite latency—the time required to complete a query rewrite—a custom Python script is utilized to invoke both rewrite methods, capturing the average rewrite latency across all test queries on the same hardware platform.

## 6.2 Executing Efficiency Evaluation

As presented in Table 2, our study conducts a comparative analysis between our proposed method **LLM-R<sup>2</sup>** and two baseline methods. We meticulously document the mean, median, 75th percentile, and 95th percentile values of execution times to provide a comprehensive performance evaluation. The mean and median offer insights into the general efficacy of the methods across the datasets, whereas the 75th and 95th percentiles facilitate an understanding of the methods’ behavior for long tail cases. Our analysis yields several key observations:

**(1) LLM-R<sup>2</sup>** demonstrates superior reduction of query execution time, outshining all baseline methods across the three datasets. Specifically on the TPC-H, IMDB and DSB datasets, **LLM-R<sup>2</sup>** reduces the execution time of the queries on average to 94.5%, 63.1% and 40.7% of the queries rewritten by baseline method **LR**, 52.7%, 56.0% and 33.1% relative to **LLM only**, and even further 52.5%, 56.0% and 39.8% compared to the original query. This performance enhancement is attributed to the optimization of demonstration selection for prompting the LLM-enhanced rewrite system, enabling **LLM-R<sup>2</sup>** to suggest superior rewrite rules. Furthermore, leveraging an LLM-enhanced system, **LLM-R<sup>2</sup>** offers more adaptable rule suggestions and better tailors these to the input query than does the **LR** baseline.

Execution time(sec)	TPC-H 1G				TPC-H 5G				TPC-H 10G			
Method	Mean	Median	75th	95th	Mean	Median	75th	95th	Mean	Median	75th	95th
Original	52.02	0.57	1.39	300.00	53.90	3.27	11.53	300.00	70.90	22.00	37.01	300.00
<b>LLM-R<sup>2</sup></b>	<b>15.19</b>	<b>0.56</b>	<b>1.14</b>	<b>55.20</b>	<b>19.34</b>	<b>3.20</b>	<b>7.97</b>	34.70	<b>37.23</b>	<b>17.40</b>	<b>29.80</b>	164.12
<b>LR</b>	25.40	0.57	<b>1.14</b>	213.81	20.10	4.02	9.02	<b>32.14</b>	39.40	22.00	37.21	<b>159.95</b>
<b>LLM only</b>	52.73	2.14	4.49	300.00	54.13	3.62	11.56	300.00	70.67	22.00	37.01	300.00

Table 6: Execution time v.s. different data scales.

(2) The margin of improvement over **LR** is notably greater in the IMDB and DSB datasets than in the TPC-H dataset. This discrepancy stems from two factors. First, TPC-H is also the mainly analysed dataset in [35] for **LR**. Most of the effective rewrite rules for TPC-H queries can already be applied by **LR**, leaving **LLM-R<sup>2</sup>** with limited scope for further enhancements. Second, the TPC-H dataset’s reliance on only 22 query templates results in a lack of query diversity, thus constraining the full demonstration of **LLM-R<sup>2</sup>**’s superiority utilizing LLM generalisation and reasoning abilities.

(3) **LR**’s under-performance in the DSB dataset can be attributed to its design limitations adopting a greedy search algorithm. The DSB dataset, being entirely new and unmet for **LR**, poses unique challenges. Moreover, the Monte Carlo tree search algorithm employed by **LR**, with its greedy search strategy that retains only a select few best options at each step, struggles with the dataset’s complex and expensive query trees. This limitation makes it difficult for the algorithm to select the most effective rules, to which explains its poor performance in handling the DSB dataset’s demands.

(4) **LLM only** has the worst performance. We observe that LLMs struggle to effectively address the query rewrite challenge, and has only marginal reductions in mean cost on the TPC-H dataset and median cost on the DSB dataset. Given that non-executable or non-equivalent rewrite attempts are categorized as ‘no rewrite,’ many rewritten queries are the same as the original queries across the datasets.

Furthermore, we evaluate the performance by collecting statistics on the number of successful rewrites performed by each method across three datasets. As shown in Table 3, we observe that:

(1) **LLM-R<sup>2</sup>** excels by having the most efficiency-enhancing rewrites, achieving the largest improvement percentage upon rewriting. Compared to the baseline, **LLM-R<sup>2</sup>** has both a higher number of rewrites and a significant improvement in query execution efficiency across all the evaluated datasets.

(2) **LLM only** fails in most of its rewrite attempts. We look into the rewrites which do not return the same results as the original queries in the TPC-H dataset, 119 of the total 129 queries are either not consistent with the original one or have errors to execute. Similarly, 193 of the 202 attempts to rewrite failed in the DSB dataset, since the DSB queries and schema would be too complicated for the LLM. This observation aligns with the results in [20], in which the text-to-SQL task only achieved around 40% accuracy with carefully designed prompts. Although the IMDB dataset is simpler compared to TPC-H and DSB datasets, where **LLM only** only fails 31 of the total 102 attempts, the LLM makes limited effective rewrites due to lack of database and query structure knowledge. In contrast, our **LLM-R<sup>2</sup>**, which benefits from both the reasoning ability of LLM and the rewrite ability of database platforms, is able to rewrite more

queries successfully and have as higher rewrite improvement rate across all the datasets.

### 6.3 Computational Efficiency Evaluation

To evaluate the computational efficiency, we rigorously assess the average rewrite latency for input queries across all datasets for the **LLM-R<sup>2</sup>** framework as well as the **LR** and **LLM only** baselines. Moreover, to ascertain if query time reduction adequately compensates for the rewriting latency, we combine the execution cost and rewrite latency to formulate a comprehensive metric. As delineated in Table 4, our analysis yields significant insights:

(1) **LLM-R<sup>2</sup>** incurs additional latency compared to **LR**, specifically requiring an average of 1.82, 1.86, and 1.51 seconds more to rewrite queries from the TPC-H, IMDB, and DSB datasets, respectively. This heightened latency is due to our system’s complexity. Notably, **LLM-R<sup>2</sup>** employs a demonstration selection model and leverages the online LLM API, which together account for the increased rewrite latency.

(2) However, the increased rewrite latency in our system **LLM-R<sup>2</sup>** is justifiable given that the sum of rewrite latency and execution time is lower than that of baseline methods, especially for the most complicated DSB queries. This indicates that the complex queries benefit more from our method.

(3) The **LLM only** approach exhibits considerable latency as the LLM endeavors to directly generate a rewritten query, underscoring the complexity of direct SQL query generation for LLMs. This latency becomes more pronounced with the complexity of the query and database, notably in the TPC-H and DSB datasets. The comparison between our **LLM-R<sup>2</sup>** framework and the **LLM only** approach demonstrates that our methodology, which focuses on generating rewrite rules, is more effectively processed by LLMs.

### 6.4 Robustness Evaluation

We next evaluate the robustness of our **LLM-R<sup>2</sup>** framework, focusing on two critical dimensions: transferability and flexibility. Transferability evaluates the system’s ability to generalize across diverse datasets, while flexibility examines whether **LLM-R<sup>2</sup>** maintains its high performance as the volume of data increases. These aspects are crucial for understanding the adaptability and efficiency of **LLM-R<sup>2</sup>** in varied environments.

**6.4.1 Transferability across different datasets.** In order to evaluate our method’s transferability, we used the demonstration selection model trained on the TPC-H dataset to rewrite queries in the IMDB dataset. As shown in Table 5, the results reveal our method’s transferred performance is comparable with the in-distribution trained method and highly superior over **LLM only** when applied to a different dataset. **LLM only** fails to make effective rewrites given the

Execution time(sec)	TPC-H				IMDB				DSB			
Method	Mean	Median	75th	95th	Mean	Median	75th	95th	Mean	Median	75th	95th
Original	70.90	22.00	37.01	300.00	6.99	1.86	5.12	32.49	60.55	6.64	26.55	300.00
<b>Zero-shot</b>	46.15	21.95	33.26	300.00	6.98	1.85	5.12	32.49	34.53	3.35	<b>11.52</b>	300.00
<b>Random</b>	40.50	21.63	32.22	165.63	5.45	1.70	4.50	25.03	45.88	5.43	17.41	300.00
<b>Tree</b>	39.21	18.97	30.89	<b>164.10</b>	4.40	<b>1.24</b>	<b>3.40</b>	18.89	26.10	3.86	13.54	240.74
<b>SentTrans</b>	40.19	19.21	32.21	164.99	6.05	1.70	4.49	30.01	24.68	3.95	13.18	197.23
<b>LLM-R<sup>2</sup></b>	<b>37.23</b>	<b>17.40</b>	<b>29.80</b>	164.12	<b>3.91</b>	1.33	3.52	<b>18.16</b>	<b>24.11</b>	<b>2.16</b>	12.61	<b>196.61</b>

Table 7: Execution time v.s. different selection approaches.

fixed demonstration from the TPC-H dataset, where most rewrites lead to meaningless changes like removing table alias. Since **LR**’s cost model lacks cross-dataset transfer capability, its results are not available. These findings suggest the potential to develop a robust model by combining multiple datasets, enhancing its ability to address a wide array of unseen queries and datasets.

**6.4.2 Flexibility across different data scales.** To further analyse the flexibility of our method, we regenerate the TPC-H dataset using different scale factors. We additionally generate TPC-H dataset using scale factor 1 (around 1GB data) and 5 (around 5GB data) apart from 10 in the main results to simulate a change of database size. From scale factor 1 to 10, we can see in Table 6 the efficiency of queries rewritten by our method increases consistently and surpasses the baseline methods.

## 6.5 Ablation Studies

We conduct an ablation study to evaluate our method’s performance along two distinct dimensions: *different selection approaches* and *specific settings in the selection model*. At first, we explore alternative selection approaches by substituting the learned selection model with different approaches to gauge their impact. Subsequently, we delve into the intricacies of the selection model by replacing individual components of the model.

**6.5.1 Different selection approaches.** We design the following approaches to replace the contrastive selection model in our system:

- **Zero-shot:** This method employs the LLM-R<sup>2</sup> to rewrite input queries without any preliminary demonstrations.

- **Few-shots:** Building on insights from Section 4, we refine the demonstration pool with three intuitive methods for one-shot demonstration selection: **Random**, **Tree**, and **SentTrans**.

Table 7 shows the results and we make the following observations:

**(1) Effectiveness of the LLM-enhanced system:** The **Zero-shot** approach outperforms the original queries significantly, which indicates that the LLM-R<sup>2</sup> component within our rewrite system is capable of enhancing original queries, showcasing the underlying potential of the LLM to offer viable query rewrite suggestions. This observation suggests that even though the recommendations provided may not always be optimal—owing to constraints such as incomplete information and occasional inaccuracies—the LLM’s contributions are valuable in improving query performance.

**(2) Effectiveness of introducing demonstrations:** We observe that approaches incorporating demonstrations into the rewrite system consistently surpass the Zero-shot setting across all datasets.

Execution time(sec)	Mean	Median	75th	95th
Original	70.90	22.00	37.01	300.00
<b>LLM-R<sup>2</sup> (1-shot)</b>	<b>37.23</b>	<b>17.40</b>	<b>29.80</b>	<b>164.12</b>
<b>w/o Curriculum</b>	38.73	19.70	32.17	164.98
<b>LLM-R<sup>2</sup> (3-shots)</b>	54.08	19.67	37.01	300.00
<b>LLM-R<sup>2</sup> (GPT-4)</b>	38.58	20.32	32.27	167.26

Table 8: Performance comparison of LLM-R<sup>2</sup> with and without the curriculum learning pipeline on TPC-H

The sole exception is observed with the **Random** method, which falls short of the Zero-shot rewrite performance on the DSB dataset. This observation underscores the significance of leveraging demonstrations to enhance the rewrite system, significantly boosting the quality of rewrites. Furthermore, the improvement across diverse datasets highlights the universal applicability and effectiveness of demonstration-based prompting in refining rewrite outcomes.

**(3) Effectiveness of the contrastive selection model:** Our comparative analysis underscores the significance of selecting high-quality demonstrations for query rewriting. The findings reveal that superior demonstrations directly contribute to the generation of more effective rewritten queries.

**6.5.2 Effectiveness of specific settings in the selection model.** In this experiment, we concentrate on assessing three critical aspects within the contrastive selection model:

- **The Curriculum Learning pipeline:** We investigate the curriculum learning pipeline’s efficacy by comparing it with a baseline model. Specifically, this baseline involves training a selection model on the TPC-H dataset using all training triplets simultaneously, rather than employing a curriculum learning-based approach.

- **Demonstration Quantity:** We evaluate the impact of varying the number of demonstrations by focusing on the most prevalent configurations—namely, 1-shot and 3-shot demonstrations. This experiment aims to elucidate the demonstration quantity’s effect on the model’s performance.

- **Different LLMs:** We explore the implications of integrating GPT-4, a more advanced LLM recognized for its superior capabilities in natural language processing, into our rewriting system. Given the financial implications of utilizing the GPT-4 API, our experimental setup restricts the use of GPT-4 to the enhancement of the test dataset rewrite process, with demonstrations and models derived from GPT-3.5-turbo.

Table 8 shows the evaluation results and we obtain the following key insights:

**(1)** Our query representation model demonstrates superior performance in selecting optimal demonstrations compared to baseline

<b>Original Query</b> Original query: select l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0 end) as high_line_count, sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1 else 0 end) as low_line_count from orders, lineitem where o_orderkey = l_orderkey and ... group by l_shipmode order by l_shipmode; Query cost: 21.63845666	<b>Original Query</b> Original query: select s_acctbal, ... , s_comment from part, supplier, ... region where p_partkey = ps_partkey and ... and ps_supplycost = ( select min(ps_supplycost) from partsupp, supplier, nation, region where p_partkey = ps_partkey and ... and r_name = 'AMERICA') order by s_acctbal desc, ... , p_partkey; Query cost: 0.789705753	<b>Original Query</b> Original query: select l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority from customer, orders, lineitem where c_mktsegment = 'AUTOMOBILE' and ... group by l_orderkey, o_orderdate, o_shippriority order by revenue desc, o_orderdate; Query cost: 33.2621007
<b>LR Result</b> Rules applied: []	<b>LR Result</b> Rules applied: ['AGGREGATE_JOIN_TRANSPOSE', 'SORT_PROJECT_TRANSPOSE', 'JOIN_EXTRACT_FILTER'] New query cost: 17.40275009	<b>LR Result</b> Rules applied: ['FILTER_INTJOIN', 'PROJECT_TO_CALC', 'JOIN_EXTRACT_FILTER'] New query cost: 33.1077284
<b>QueryCL Result</b> Rules applied: ['FILTER_INTJOIN'] New query cost: 17.65797496	<b>QueryCL Result</b> Rules applied: []	<b>QueryCL Result</b> Rules applied: ['FILTER_INTJOIN', 'JOIN_EXTRACT_FILTER', 'PROJECT_TO_CALC', 'FILTER_INTJOIN'] New query cost: 26.90622425

Figure 8: Examples of the rewrite results of baseline Learned Rewrite method and out LLM-R<sup>2</sup> method.

approaches, and the incorporation of a curriculum-based training methodology significantly amplifies this advantage. For instance, direct training on the complete dataset results in a notable reduction in execution cost, averaging a decrease of 32.17 seconds and a median of 2.3 seconds, respectively. Utilizing the curriculum learning approach for training the demonstration selector further contributes to cost efficiency, achieving an average reduction of 1.5 seconds and a median decrease of 2.3 seconds. These findings underscore the efficacy of our proposed query representation model and the curriculum learning framework.

(2) Employing a 3-shot approach, as opposed to a 1-shot strategy, adversely affects performance. A detailed examination of the rewritten queries reveals that, the 3-shot method generated only 255 rewrite proposals, and 235 of these rewrites yielded improvements in query execution efficiency. Despite a high success rate of 92.16% for these rewrites, the primary limitation lies in the significantly reduced number of rewrite suggestions. This reduction is largely attributed to the inconsistent guidance provided by the three demonstrations. Additionally, the increased cost of rewrites and the challenges posed by longer in-context texts for LLM analysis emerge as critical yet unresolved issues when employing 3-shot prompting. Based on these findings, we deduce that 1-shot prompting presents a more efficient and effective approach under the current experimental conditions.

(3) Despite GPT-4’s enhanced capabilities, transitioning to a different model for inference adversely impacts the efficacy of our method. This observation underscores the complexity of optimizing performance within our proposed framework and suggests that consistency in model usage throughout the process may be pivotal for achieving optimal selection.

## 6.6 Qualitative Analysis

we proceed to present examples to illustrate the rewrite quality between various methods, focusing particularly on comparisons between our approach and baseline methods. Notably, due to the high incidence of erroneous rewrites generated by the **LLM-only** method, our analysis primarily compares our method against the **LR** baseline. Figure 8 demonstrates our findings demonstrate the superior robustness and flexibility of our model compared to **LR**. For instance, in the first case study, our **LLM-R<sup>2</sup>** method uncovers rewrite rules that remain undetected by **LR**. This discrepancy can be attributed to **LR**’s potentially ineffective cost model, which might erroneously consider the original query as already optimized. Conversely, our LLM-enhanced system suggests a rewrite that evidences significant potential for cost reduction. In the second case, **LR** is observed to occasionally transform an efficient query into a less efficient one. In the third scenario, **LLM-R<sup>2</sup>** outperforms by modifying the rule sequence and incorporating an additional

Counts	TPC-H		IMDB		DSB	
Method	Unique	Total	Unique	Total	Unique	Total
LR	5	405	1	192	9	707
LLM-R <sup>2</sup>	56	1824	6	361	37	920

Table 9: The variety of rules applied by the methods in terms of unique rules and total applications.

“FILTER\_INTJOIN” operation, transforming a “WHERE” clause into an “INNER JOIN”, thereby achieving a more efficient query rewrite than that offered by **LR**.

Furthermore, we delve into the diversity of rewrite rules suggested by the different methods. Here, the term *Unique* refers to the distinct categories of rewrite rules recommended by a method, whereas *Total* denotes the aggregate count of all rewrite rule instances proposed. As illustrated in Table 9, it is evident that **LLM-R<sup>2</sup>** not only recommends a higher quantity of rewrite rules but also exhibits a broader spectrum of rewrite strategies by employing a diverse range of rules. This observation underscores **LLM-R<sup>2</sup>**’s enhanced flexibility and robustness, showcasing its capability to generate more varied and effective rewrite plans.

## 7 CONCLUSION

Despite the analysis above, we would like to point out the current limitation for further work. The main limitation for our **LLM-R<sup>2</sup>** lies in the higher rewrite latency compared to DB only methods. Compared to traditional DB methods, calling LLM API and selecting demonstrations indeed consume more time. However, as shown in the experiment results, such higher latency can be alleviated by the larger execution time **LLM-R<sup>2</sup>** decreases, and there is no doubt that our **LLM-R<sup>2</sup>** is a successful example of exploring the LLMs’ application in database problems. We believe that the strong generalisation and reasoning ability of the LLMs can also be applied to other important database problems as well. In addition, further work can also be made to improve our current LLM enhanced query rewrite system, for example, utilising efficient demonstration selection algorithms like Faiss [12], or even specially fine-tune a LLM on query rewrite with more dataset.

To conclude, we propose a LLM-enhanced query rewrite pipeline to perform efficient query rewrite. By collecting useful demonstrations and learning a contrastive demonstration selector to modify the rewrite system inputs, we are able to successfully improve the input queries’ efficiency across popular datasets. In addition, we further prove the effectiveness of our learning pipeline and the transferability of our method over different scales, model backbones and datasets, showing that LLM-enhanced methods could be an effective solution for efficiency-oriented query rewrite.

## REFERENCES

- [1] [n.d.]. Apache Calcite Rewrite Rules. <https://calcite.apache.org/javadocAggregate/org/apache/calcite/rel/rules/package-summary.html>.
- [2] [n.d.]. Introduction of OpenAI Text Generation APIs. <https://platform.openai.com/docs/guides/text-generation>.
- [3] [n.d.]. LLM As Database Administrator. <https://github.com/TsinghuaDatabaseGroup/DB-GPT>.
- [4] [n.d.]. PostgreSQL. <https://www.postgresql.org>.
- [5] [n.d.]. TPC-H Toolkit. [https://www.tpc.org/tpc\\_documents\\_current\\_versions/current\\_specifications5.asp](https://www.tpc.org/tpc_documents_current_versions/current_specifications5.asp).
- [6] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. *arXiv:2305.08272* [cs.DB]
- [7] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD/PODS '18)*. ACM. <https://doi.org/10.1145/3183713.3190662>
- [8] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning (Montreal, Quebec, Canada) (ICML '09)*. Association for Computing Machinery, New York, NY, USA, 41–48. <https://doi.org/10.1145/1553374.1553380>
- [9] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *arXiv:2005.14165* [cs.CL]
- [10] Daniel Cer, Mona Diab, Eneko Agirre, Inigo Lopez-Gazpio, and Lucia Specia. 2017. SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*. Association for Computational Linguistics. <https://doi.org/10.18653/v1/s17-2001>
- [11] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. In *VLDB 2022*. <https://www.microsoft.com/en-us/research/publication/dsb-a-decision-support-benchmark-for-workload-driven-and-traditional-database-systems/>
- [12] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). *arXiv:2401.08281* [cs.LG]
- [13] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2022. SimCSE: Simple Contrastive Learning of Sentence Embeddings. *arXiv:2104.08821* [cs.CL]
- [14] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data(base) Engineering Bulletin* 18 (1995), 19–29. <https://api.semanticscholar.org/CorpusID:260706023>
- [15] Goetz Graefe and David J. DeWitt. 1987. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (San Francisco, California, USA) (SIGMOD '87)*. Association for Computing Machinery, New York, NY, USA, 160–172. <https://doi.org/10.1145/38713.38734>
- [16] G. Graefe and W.J. McKenna. 1993. The Volcano optimizer generator: extensibility and efficient search. In *Proceedings of IEEE 9th International Conference on Data Engineering*. 209–218. <https://doi.org/10.1109/ICDE.1993.344061>
- [17] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *Comput. Surveys* 55, 12 (March 2023), 1–38. <https://doi.org/10.1145/3571730>
- [18] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9 (2015), 204–215. <https://api.semanticscholar.org/CorpusID:7953847>
- [19] Feifei Li. 2019. Cloud-Native Database Systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (aug 2019), 2263–2272. <https://doi.org/10.14778/3352063.3352141>
- [20] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C. C. Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A Blg Bench for Large-Scale Database Grounded Text-to-SQLs. *arXiv:2305.03111* [cs.CL]
- [21] Xiaonan Li, Kai Lv, Hang Yan, Tianyang Lin, Wei Zhu, Yuan Ni, Guotong Xie, Xiaoling Wang, and Xipeng Qiu. 2023. Unified Demonstration Retriever for In-Context Learning. *arXiv:2305.04320* [cs.CL]
- [22] Andrew L. Maas, Raymond E. Daly, Peter T. Pham, Dan Huang, Andrew Y. Ng, and Christopher Potts. 2011. Learning Word Vectors for Sentiment Analysis. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Portland, Oregon, USA, 142–150. <http://www.aclweb.org/anthology/P11-1015>
- [23] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. *SIGMOD Rec.* 21, 2 (jun 1992), 39–48. <https://doi.org/10.1145/141484.130294>
- [24] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. <https://arxiv.org/abs/1908.10084>
- [25] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. *arXiv:2302.04761* [cs.CL]
- [26] Ruoxi Sun, Serkan O. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *arXiv:2306.00739* [cs.CL]
- [27] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *Proceedings of the 2022 International Conference on Management of Data (Philadelphia, PA, USA) (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 94–107. <https://doi.org/10.1145/3514221.3526125>
- [28] Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, and Tengyu Ma. 2023. Larger language models do in-context learning differently. *arXiv:2303.03846* [cs.CL]
- [29] Wentao Wu, Philip A. Bernstein, Alex Raizman, and Christina Pavlopoulou. 2022. Factor Windows: Cost-based Query Rewriting for Optimizing Correlated Window Aggregates. *arXiv:2008.12379* [cs.DB]
- [30] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Danrui Qi, Hong Yi, Shaocong Liu, and Faqiang Chen. 2024. DB-GPT: Empowering Database Interactions with Private Large Language Models. *arXiv:2312.17449* [cs.DB]
- [31] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. *arXiv:2210.03629* [cs.CL]
- [32] Yue Zhang, Yafu Li, Leyang Cui, Deng Cai, Lemao Liu, Tingchen Fu, Xinting Huang, Enbo Zhao, Yu Zhang, Yulong Chen, Longyue Wang, Anh Tuan Luu, Wei Bi, Freda Shi, and Shuming Shi. 2023. Siren’s Song in the AI Ocean: A Survey on Hallucination in Large Language Models. *arXiv:2309.01219* [cs.CL]
- [33] 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 (04 2022), 1658–1670. <https://doi.org/10.14778/3529337.3529349>
- [34] Yue Zhao, Zhaodonghui Li, and Gao Cong. 2024. A Comparative Study and Component Analysis of Query Plan Representation Techniques in ML4DB Studies. *Proc. VLDB Endow.* 17, 4 (2024).
- [35] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System Using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (sep 2021), 46–58. <https://doi.org/10.14778/3485450.3485456>
- [36] Yuhang Zhou, He Yu, Siyu Tian, Dan Chen, Liuzhi Zhou, Xinlin Yu, Chuanjun Ji, Sen Liu, Guangnan Ye, and Hongfeng Chai. 2023. R<sup>3</sup>-NL2GQL: A Hybrid Models Approach for Accuracy Enhancing and Hallucinations Mitigation. *arXiv:2311.01862* [cs.CL]