LLM-R^2: 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 improve query efficiency by altering an SQL query's structure without changing its 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. First, 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. Second, 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 query rewrite method named LLM-R^2, which leverages a large language model (LLM) to recommend rewrite rules for a database rewrite system. To further enhance the inference ability of the LLM in recommending rewrite rules, we train a contrastive model using a curriculum-based approach to learn query representations and select effective query demonstrations for the LLM. Experimental results show that our method significantly improves the query execution efficiency and outperforms the baseline methods. In addition, our method exhibits high robustness across different datasets.

PVLDB Reference Format:
Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. LLM-R^2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. PVLDB, 18(1): 53 - 65, 2024.
doi:10.14778/3696435.3696440

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

*Zhaodonghui Li is under the Joint PhD Program between DAMO Academy and Nanyang Technological University
†Haitao Yuan is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 1 ISSN 2150-8097.
doi:10.14778/3696435.3696440

1 INTRODUCTION

Efficient query processing has been a crucial task in modern database systems. One of the key topics in query optimization is query rewrite [22, 27]. The objective of query rewrite is to generate a new query that is equivalent to the original SQL query but executes in less time. Ideally, query rewrite should meet three critical criteria: (1) **Executability**: the rewritten query should be executable and without any errors; (2) **Equivalence**: it must produce identical results as the original query; (3) **Efficiency**: this encompasses two aspects—*Execution Efficiency* and *Computational Efficiency*. *Execution Efficiency* requires 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 saved during query execution.

To improve both **Executability** and **Equivalence** in rewritten queries, existing studies have mainly focused on rule-based rewriting techniques. In particular, these studies are divided into two complementary research directions: discovering novel rewriting rules and effectively applying existing ones. For the first direction, although additional rewrite rules [7, 34, 36] have been discovered, many challenges remains, particularly concerning the complexity of rule validation and the specificity of their applicability. These challenges often lead to high computational demands and require professional-level user competence. For example, Wetune [34] only supports discovering rewrite rules on limited types of operators and Querybooster [7] necessitates user engagement with specialized rule syntax. This work focuses on the latter direction, exploring methodologies for the effective utilization of pre-established rules. For example, Learned Rewrite [45] utilizes existing rewrite rules from the Apache Calcite [8] platform and learns to select rules to apply. It notably incorporates a Monte Carlo search algorithm together with a machine-learned query cost estimator to streamline the selection process. However, it is non-trivial to solve the challenges related to the computational demand of the Monte Carlo algorithm and the accuracy of the cost estimation model, which can significantly impact the **execution efficiency**.

On the other hand, with the rise of large language models (LLMs), several "large language model for database" projects [3, 37] have emerged, which 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 [20, 41], 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 address the limitations of the current query rewriting techniques while benefiting from their advantages, we propose an LLM-enhanced rewrite system. This system uses LLMs to recommend rewrite rules and apply these rules with an existing database platform to rewrite the input query. Inspired by the LLM-based learning framework for using tools [29, 38], we leverage the LLM's 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. Note that our approach guarantees the **executability** and **equivalence** of the rewritten query 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 **execution 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 develop 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 selection on rewrite rules. In addition, to address the challenge of limited training data, we propose using the learning curriculum technique [9] to train the model using training data in an easy to hard way. 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, requiring 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, this is the first work on 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.
- We develop a contrastive query representation model to optimize the demonstration selection. To address the challenge of limited training data, we further design a learning curriculum to organize the training data from easy to hard.
- We 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.

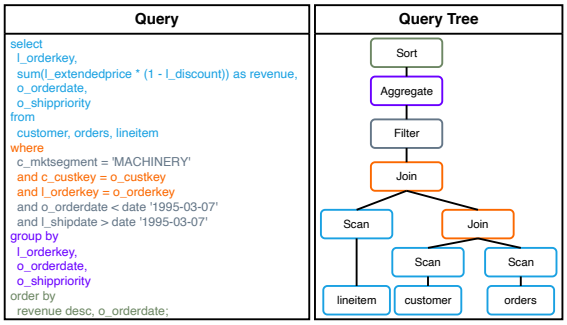

Figure 1: A TPC-H query and its query tree

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 $r_1, r_2, \cdots$, 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 $R = [r_1, r_2, \cdots, r_n]$ 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 $R^* = [r_1^*, r_2^*, \cdots, r_n^*]$ where $r_i^* \in R$, 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.

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

The problem can be formally represented as:

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

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 [16–19, 40]. In particular, DBMSs, such as Calcite [8] 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 [7] and Wetune [34], have made significant strides in discovering new rewrite rules. Querybooster enables database users to suggest rules through a specialized rule language. 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. Learned Rewrite [45] 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 Learned Rewrite 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. These models have demonstrated a surprisingly strong ability to handle a variety of text-related tasks such as generation, decision-making, and reasoning. One such task that is highly related to DB research is text-to-SQL, in which an LLM directly generates an SQL query given database information and user requirements. Numerous studies [23, 31, 46] 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 efficiency and accuracy of these queries. In particular, [23] 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 methods face is the problem of hallucination, which refers to instances where the model generates incorrect outputs but is done so with a misleading level of confidence. Although some methods try to utilize instruction tuning to solve the problem, there are still three main limitations. First, only open-source LLMs like Llama and Phi can be fine-tuned, but they lag behind closed-source LLMs like the GPT family. Second, fine-tuning techniques still cannot eliminate hallucinations. One example is the SOTA LLM for txt-to-SQL task Granite-20b-code model [26], which is an LLM specially fine-tuned on txt-to-SQL data but only 67.86% of the queries generated by the model on the BIRD benchmark [23] are executable. This is particularly problematic in the context of database applications, where accuracy is paramount and it motivates us to use rule-based methods to ensure query correctness. Lastly, leveraging LLMs' generalization ability is crucial. Instruction tuning requires re-tuning with new datasets and queries, which is inefficient. Therefore, we propose a different direction of utilising the LLMs while overcoming hallucination by using LLM's in-context learning capability and adopt 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 in-context learning (ICL). The concept of ICL, first introduced by Brown et al. in their seminal work on GPT-3 [10], 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 [35] and [24], 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.

2.3.4 Contrastive Learning by Curriculum. Contrastive learning by curriculum merges the strengths of contrastive learning and curriculum learning to create efficient machine learning models with minimal labeled data. Contrastive learning enhances representation by bringing similar data points closer and separating dissimilar ones, while curriculum learning structures the training process progressively. In particular, the SOTA method [39] in natural language processing creates data with different levels of difficulties using the PCA jittering method to form the learning curriculum. However, PCA jittering is a method that generates textual sentences and cannot be applied to generating SQL queries. Similarly, in computer vision, the SOTA methods [11] and [33] demonstrate how a curriculum can be set up to incrementally learn a classification model. They use the curriculum to incrementally train models by starting

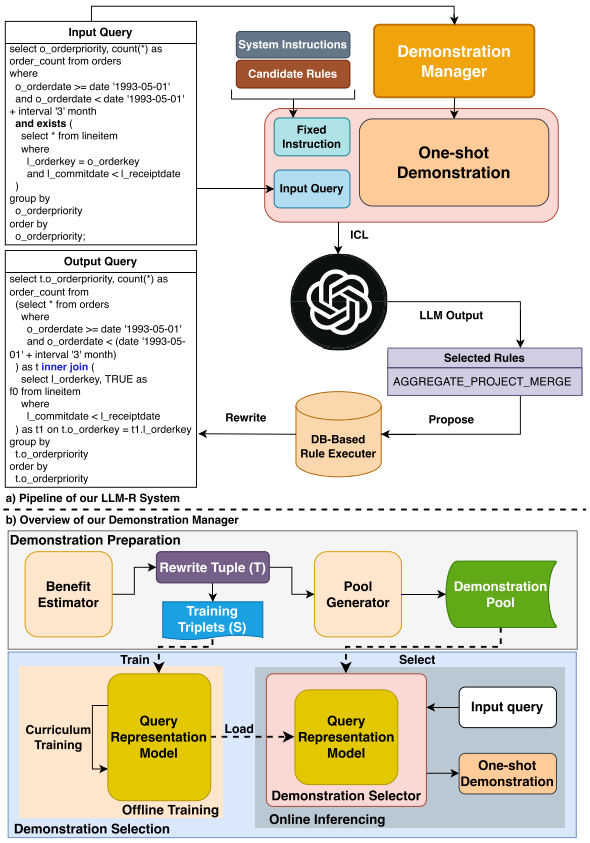

Figure 2: The Framework of LLM-enhanced Rewrite System

to focus on more confidently labeled data. However, they assume a semi-supervised setting where a computer vision model generating image pseudo-labels is required. Therefore, the SOTA methods may not directly apply to our problem and we need to adapt the idea of contrastive learning by curriculum to our own problem.

3 LLM-ENHANCED REWRITE SYSTEM

In this section, we will introduce our innovative LLM-enhanced rule-based rewrite system (**LLM-R^2**). 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 [10]. 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 letting LLMs 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^2 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 $R^D = [r_1^D, \cdots]$ 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., [10, 35]) 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

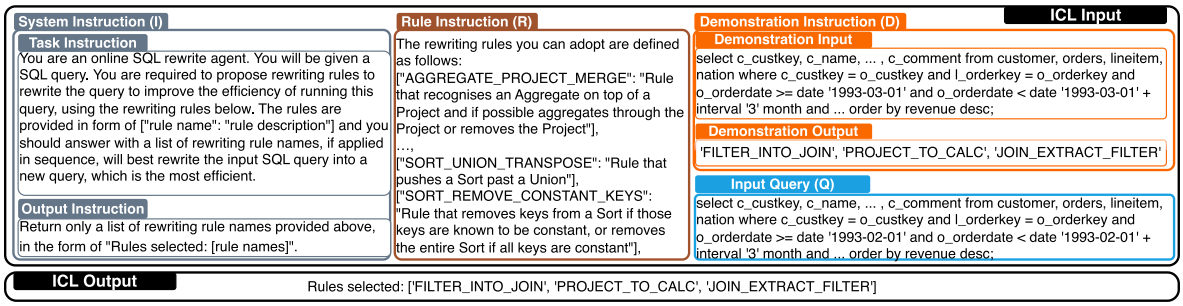

Figure 3: An Example of the In-Context Learning Process in LLM-R^2. All the instructions are concatenated together as one string input to the LLM. In a zero-shot setting, the "Demonstration Instruction" will be removed and an input query will be appended directly after the "Rule Instruction".

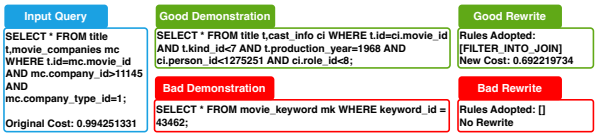

Figure 4: Example of good and bad demonstration selections

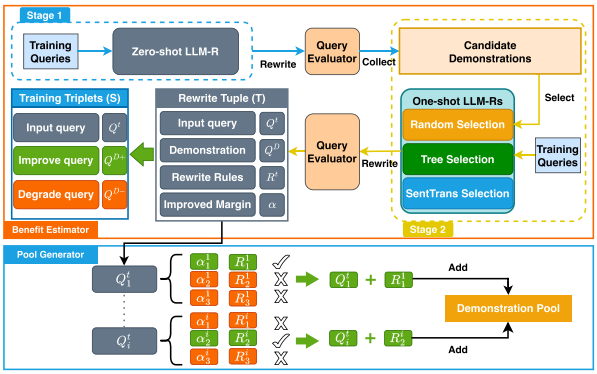

Figure 5: Our demonstration preparation module generates a set of training triplets and a demonstration pool.

the demonstration selected for a given input query. Particularly, 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 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^2** system where no demonstration is selected, followed by the demonstration adoption stage employing a one-shot **LLM-R^2** 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^2) 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

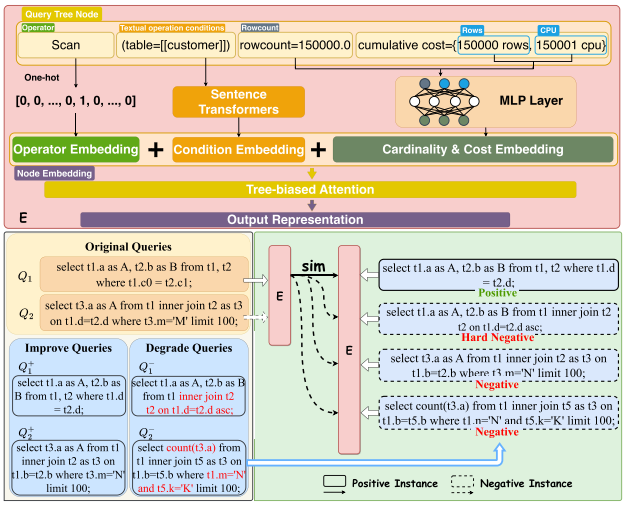

Figure 6: Our representation model encodes each query tree node into a fixed-length vector, with the final query representation obtained through tree-biased attention over the nodes. The model E is trained using contrastive query tuples.

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 [35], 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 [28], we can obtain an embedding of a given sentence.

With the three demonstration selection methods above, we can prompt our **LLM-R^2** 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^2** 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 (Q^t, D, R^t, α), 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 α 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 (Q^t, D, R^t, α), 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 $\{T_i\}_{1 \le i \le n}$ based on their corresponding input queries. Therefore, each group T_i can be represented as the tuple set $\{(Q_i^t, D_1^i, R_1^i, \alpha_1^i), (Q_i^t, D_2^i, R_2^i, \alpha_2^i), \cdots\}$. 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:

$$\begin{aligned} * &= argmax_{j \in [1, |T_i|]} \alpha_j^i \\ s.t. T_i &= \{(Q_i^t, D_1^i, R_1^i, \alpha_1^i), (Q_i^t, D_2^i, R_2^i, \alpha_2^i), \cdots\} \end{aligned} \tag{2}$$

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 α_1^1 and α_2^i are identified for input queries Q_1^t and Q_i^t, the corresponding

demonstrations $\langle Q_1^t, R_1^1 \rangle$ and $\langle Q_i^t, R_2^i \rangle$ are selected with the rewrite rules R_1^1 and R_2^i 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 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 [42] to take the query trees generated by DBMS' query analyzer as inputs. It is notable that the original encoding in [42] 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 refer to [44] and 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 [42], such information has no unified form for encoding. We consider the conditions as text and encode using a pre-trained Sentence Transformers encoder [28]. Such an encoder can capture the textual differences between conditions effectively and have unified embedding dimensions to simplify further analysis.

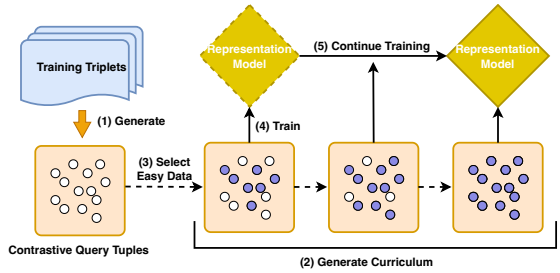

Figure 7: The overall curriculum learning pipeline to train the contrastive selector using generated training triplets.

- **Cardinality and cost:** From [43] 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 [42] 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 [42], 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 [15], 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

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

where τ 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.

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 [12], 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.

Algorithm 1 Contrastive Training under Curriculum Scheduler

Require: Total training data S_0, Number of iterations I
Require: Initialized model E_0
1: $N_0 = len(S_0)$ ▷ 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** $len(N) > len(S_{i-1})$ **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 training
10: **else**
11: $C_i \leftarrow Top_N(S_{i-1})$ based on $conf_{E_{i-1}}(\cdot)$ ▷ 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

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:

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

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) [21]: The IMDB [25] 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) [13]: 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 [8] as our rewrite platform, alongside its comprehensive set of rewrite rules by following previous work [45]. 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.

6.1.3 LLM Setting. We leverage the GPT-3.5-turbo version [10] within the ChatGPT API [2] as the default LLM setting. Furthermore, we assess our system's generalizability across other LLMs (e.g., the leading closed-source model GPT-4 and the leading open-source models Llama3 and Granite), as detailed in Section 6.5.

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

Learned Rewrite (LR) [45]: 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 [23]: 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 use the original queries to ensure a fair comparison.

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

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

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
LR	39.40	22.00	32.21	**159.95**	6.20	1.62	4.74	32.45	59.21	5.14	53.78	300.00
LLM only (GPT-3.5)	70.67	22.00	37.01	300.00	6.96	1.86	5.10	32.49	61.60	6.53	26.40	300.00
LLM-R^2 (GPT-3.5)	**37.23**	**17.40**	**29.80**	164.12	**3.91**	**1.33**	**3.52**	**18.16**	**24.11**	**2.16**	**12.61**	**196.61**
% of **Original**	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 **LR**	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 **LLM only**	52.7%	79.1%	80.5%	54.7%	56.2%	71.3%	69.0%	55.9%	39.1%	33.1%	47.8%	65.5%
LLM only (Llama3)	70.80	22.00	37.01	300.00	6.98	1.86	4.99	32.49	62.02	6.62	26.55	300.00
LLM only (Granite)	65.06	20.73	36.25	300.00	6.03	1.85	4.91	32.49	60.55	6.64	26.55	300.00
LLM-R^2 (Llama3)	38.47	**18.99**	31.55	161.03	5.94	1.83	4.88	29.69	51.35	6.12	27.07	**300.00**
LLM-R^2 (Granite)	**37.89**	19.75	**28.62**	**157.37**	**4.71**	**0.97**	**3.65**	**21.56**	**26.58**	**3.46**	**13.15**	**300.00**

Table 3: The rewritten queries' number

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

Table 4: Query execution time including the rewrite latency

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

Table 5: The average monetary cost, average number of tokens and time cost of training LLM-R^2.

Method	Dataset	Avg_Cost	Avg_Tk._In	Avg_Tk._Out	Training_T.
LLM Only	TPC-H	0.0017	2844.16	161.54	-
	IMDB	0.0012	1371.36	49.76	-
	DSB	0.0028	5571.82	417.73	-
LLM-R^2	TPC-H	**0.0006**	**1167.5**	**20.31**	4061
	IMDB	**0.0006**	**1188.36**	**21.35**	3305
	DSB	**0.0008**	**1299.67**	**7.00**	6571

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 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 shown in Table 2, we compare our proposed method **LLM-R^2** with two baselines, documenting the mean, median, 75th, and 95th percentile execution times. The mean and median indicate general efficacy, while the 75th and 95th percentiles highlight performance in long tail cases. Our analysis reveals several key observations:
(1) LLM-R^2 significantly reduces query execution time across the TPC-H, IMDB, and DSB datasets, outperforming all baseline methods. Specifically, **LLM-R^2** reduces execution time to 94.5%, 63.1%, and 40.7% compared to **LR**, and to 52.7%, 56.0%, and 33.1% relative to **LLM only**, with further reductions to 52.5%, 56.0%, and 39.8% compared to the original query. This performance enhancement is due to optimized demonstration selection for prompting the LLM's input, allowing our method to suggest superior rewrite rules. Additionally, **LLM-R^2** offers more adaptable and tailored rule suggestions compared to the **LR** baseline.
(2) The bottom section of Table 2 indicates that **LLM-R^2** using different LLM backbones can also outperform both **LLM Only** and **LR**, showcasing the effectiveness of our **LLM-R^2** for both closed-source and open-source LLMs.
(3) The margin of improvement over **LR** is significantly greater in the IMDB and DSB datasets compared to the TPC-H dataset. This is due to two reasons: (1) Most of the effective rewrite rules for TPC-H queries can already be applied by existing methods, limiting **LLM-R^2**'s potential for enhancements; (2) TPC-H's reliance on only 22 query templates results in limited query diversity, constraining the demonstration of **LLM-R^2**'s superior generalization abilities.
(4) LR's underperformance on the DSB dataset is due to its greedy search algorithm. The Monte Carlo tree search in **LR** struggles with the complex and costly query trees of DSB, retaining only a few best options at each step. This limitation hampers the selection of effective rules, explaining its poor performance.
(5) LLM only has the worst performance. The direct generation of SQL with LLMs results in non-executable or non-equivalent rewrites, and hence many rewritten queries remain identical to the original across 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^2 excels with the most efficiency-enhancing rewrites, achieving the largest improvement percentage upon rewriting. Compared to the baseline, **LLM-R^2** has both a higher number of rewrites and a significant improvement in query execution efficiency across all the evaluated datasets.
(2) LLM only often fails in its rewrite attempts. For instance, in the TPC-H dataset, 119 out of 129 rewrites either do not match the original query results or contain execution errors. Even in the simpler IMDB dataset, **LLM only** fails in 31 out of 102 attempts and makes limited effective rewrites due to a lack of database knowledge. In contrast, our **LLM-R^2** successfully rewrites more queries and achieves a higher improvement rate across all 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^2** 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

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

Execution time(sec)	TPC-H 1G				TPC-H 5G				TPC-H 10G				TPC-H 100G			
Method	Mean	Median	75th	95th	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	1296.85	54.35	3000.00	3000.00
LLM-R^2	**15.19**	**0.56**	**1.14**	**55.20**	**19.34**	**3.20**	**7.97**	34.70	**37.23**	**17.40**	**29.80**	164.12	**963.13**	**32.60**	**1330.72**	**3000.00**
LR	25.40	0.57	**1.14**	213.81	20.10	4.02	9.02	**32.14**	39.40	22.00	37.21	**159.95**	1037.02	44.68	2265.75	3000.00
LLM only	52.73	2.14	4.49	300.00	54.13	3.62	11.56	300.00	70.67	22.00	37.01	300.00	1304.37	54.15	3000.00	3000.00

rewrite latency to formulate a comprehensive metric in Table 4. In order to further analyse the efficiency of **LLM-R^2**, we also compute the monetary cost and **LLM-R^2**'s model training time in Table 5. From the evaluations, our analysis yields significant insights:

(1) LLM-R^2 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^2** 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^2** 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. Since we have included information like table and cardinality in the demonstration selection module, **LLM-R^2**'s input to LLM is much shorter than that of **LLM only**, where we also need to include the basic information like schema and query's execution plan as inputs. The details of the input and output length difference can be found in Table 5. We also observe that this difference 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^2** framework and the **LLM only** approach demonstrates that our methodology, which focuses on generating rewrite rules, is more effectively processed by LLMs.

(4) We also present the monetary cost and training time of our method using one Tesla-V100-16G GPU on all three datasets in seconds. As shown in Table 5, we observe that the average monetary cost per query for our **LLM-R^2** is less than 0.001 USD for all datasets, which can be considered to be cost-efficient even processing large number of queries. In addition, using **LLM only** takes around 2 to 3 times of **LLM-R^2**'s monetary cost. Although the average cost per query for **LLM only** is still not too high, the efficiency for using LLMs along is still much lower than our method. Moreover, training our model does not cost too much time. A single time training for less than two hours will ensure the model to be capable of dealing with all kinds of input queries querying the database.

6.4 Robustness Evaluation

We next evaluate the robustness of our **LLM-R^2** 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^2** maintains its high performance as the volume of data increases. These aspects are crucial for understanding the adaptability and efficiency of **LLM-R^2** in varied environments.

Table 7: Training on TPC-H and Testing on IMDB

Execution time(sec)	Mean	Median	75th	95th
Original	6.99	1.86	5.12	32.49
LLM-R^2	**4.41**	**1.35**	**3.57**	**17.84**
LR	-	-	-	-
LLM only	6.99	1.86	5.12	32.49

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 7, 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 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 analyze 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), 5 (around 5GB data) and 100 (around 100GB data) apart from 10 in the main results to simulate a change of database size. From scale factor 1 to 100, we can see in Table 6 the efficiency of queries rewritten by our method increases consistently and surpasses the baseline methods, indicating the overall efficacy of our method.

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^2 to rewrite input queries without any preliminary demonstrations.

Table 8: Execution time v.s. different selection approaches.

Execution time	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
Zero-shot	46.15	21.95	33.26	300.00	6.98	1.85	5.12	32.49	34.53	3.35	**11.52**	300.00
Random	40.50	21.63	32.22	165.63	5.45	1.70	4.50	25.03	45.88	5.43	17.41	300.00
Tree	39.21	18.97	30.89	**164.10**	4.40	**1.24**	**3.40**	18.89	26.10	3.86	13.54	240.74
SentTrans	40.19	19.21	32.21	164.99	6.05	1.70	4.49	30.01	24.68	3.95	13.18	197.23
LLM-R^2	**37.23**	**17.40**	**29.80**	164.12	**3.91**	1.33	3.52	**18.16**	**24.11**	**2.16**	12.61	**196.61**

Table 9: Ablation on curriculum, number of shots and LLM backbone.

Execution time(sec)	Mean	Median	75th	95th
LLM-R^2 (1-shot)	**37.23**	**17.40**	29.80	164.12
w/o Curriculum	38.73	19.70	32.17	164.98
LLM-R^2 (3-shots)	54.08	19.67	37.01	300.00
LLM-R^2 (GPT-4)	38.58	20.32	32.27	167.26
LLM-R^2 (Llama3)	38.47	18.99	31.55	161.03
LLM-R^2 (Granite)	37.89	19.75	**28.62**	**157.37**

- **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**.

As shown in Table 8 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^2 component within our rewrite system is capable of enhancing original queries, showcasing the underlying potential of the LLM to offer viable query rewrite suggestions.

(2) Effectiveness of introducing demonstrations: We observe that approaches incorporating demonstrations into the rewrite system consistently surpass the Zero-shot setting across all datasets. This observation underscores the significance of leveraging demonstrations to enhance the rewrite system. 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 assess three critical aspects of the contrastive selection model:

- **The Curriculum Learning pipeline:** We compare the efficacy of the curriculum learning pipeline with a baseline model trained on the TPC-H dataset using all training triplets simultaneously.
- **Demonstration Quantity:** We evaluate the impact of varying the number of demonstrations, focusing on 1-shot and 3-shot configurations to understand their effect on model performance.
- **Different LLMs:** We explore the integration of GPT-4, Llama3-8B [32] and Granite [26] into our rewriting system. Due to the cost of the GPT-4 API, we limit its use to the test dataset rewrite process, using GPT-3.5-turbo for demonstrations and models.

Table 9 shows the evaluation results and we obtain the following key insights:

(1) Our query representation model outperforms the baseline approaches in selecting optimal demonstrations, especially when curriculum-based training is adopted. Direct training on the full dataset reduces execution costs by an average of 32.17 seconds and a median of 2.3 seconds. Curriculum learning further enhances efficiency, with average reductions of 1.5 seconds and median decreases of 2.3 seconds. These results underscore the efficacy of our proposed query representation model and the curriculum learning framework.

Table 10: Ablation on different information factors sorted by Mean value.

TPC-H	Execution time	Counts
Method	Mean/Median/75th/95th	Total/ Improve/%
LLM-R^2	**37.23**/**17.40**/**29.80**/164.12	323/305/94.43
LR	39.40/22.00/32.21/**159.95**	258/192/74.42
Full_schema+Value	**38.09**/**19.08**/**30.82**/**160.19**	334/313/93.71
Filtered_schema	50.36/1.27/32.66/300.00	178/155/87.08
Full_schema	50.52/20.56/32.61/300.00	179/158/88.27
Filtered_schema+Value	53.81/20.73/31.53/300.00	222/198/89.19
Full_schema+Value+Plan	54.93/22.00/37.01/300.00	135/121/89.63
Full_schema+Plan	55.24/22.00/37.01/300.00	119/93/78.15
Full_schema+Card	56.23/22.00/37.01/300.00	56/33/58.93
Plan	57.20/22.00/7.01/300.00	44/42/95.45
Full_schema+Value+Card	68.67/22.00/37.01/300.00	48/43/89.58
Card	70.90/22.00/37.01/300.00	0/0/-

(2) Using a 3-shot approach instead of 1-shot degrades performance. The 3-shot method generated only 255 rewrite proposals, with 235 improving query execution efficiency, despite a 92.16% success rate. The reduced number of suggestions is due to inconsistent guidance from three demonstrations, higher rewrite costs, and longer in-context texts for LLM analysis. Thus, 1-shot prompting is more efficient and effective under current conditions.

(3) we use the leading Llama3 from the Llama family [32] and Granite [26] as the LLM backbones for both our LLM-R^2 and the baseline LLM Only method. We evaluate them on all three datasets and record the results in Table 2. We observe that changing the LLM backbone into Llama3 or Granite decreases both LLM-R^2's and LLM Only's performance. Moreover, the performance of LLM-R^2 using Llama3 and Granite still outperforms all the baselines, showcasing the effectiveness of our method.

(4) Despite GPT-4's enhanced capabilities and Llama3 or Granite's strong instruction-following ability, 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.

6.5.3 Additional Information in LLM Prompts. Including data distribution and cardinality factors in demonstrations and the inputs of LLM is also worth discussing as they are widely adopted in txt-to-SQL methods [6, 23]. We design the following information that can be included in demonstrations and the LLM's inputs together with the queries as another ablation study:

Full_schema: The information of the database schema, including table names, column names and column types;

Filtered_schema: The filtered information of the database schema, where only the tables and columns queried are included;

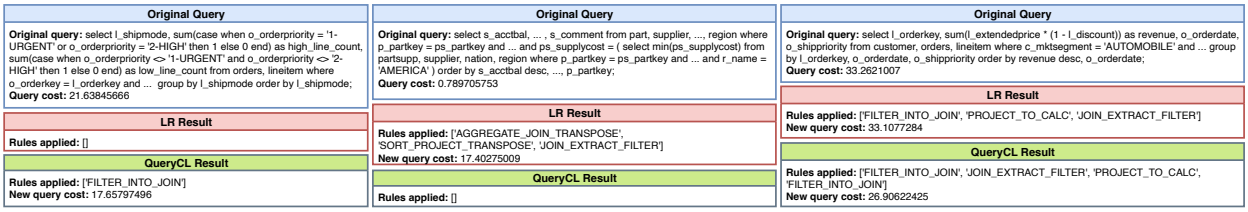

Figure 8: Examples of the rewrite results of baseline Learned Rewrite method and out LLM-R^2 method.

Table 11: The variety of rules in terms of unique rules and total applications.

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

Value: To let the LLMs understand the data distribution and value range information, we follow existing txt-to-SQL work [6, 23] to include value descriptions to each column of the database schema;
Plan: We directly include the physical plan obtained from DBMS on the query as a query tree ;
Card: We specially extract the cardinality information from the physical plans and formalize a cardinality tree as input.
The results of the experiment are shown in Table 10. From the results we have a few observations.
(1) Adding more information to demonstrations and LLM inputs does not help the LLM generate better rewrite suggestions. Our LLM-R^2 outperforms all the other combination of additional information. We believe that the Occam's Razor also exists in LLMs, especially for dealing with complex task like SQL.
(2) Current LLMs still cannot handle structural information well when directly given such information. When we include Plan or Card in the inputs, which are in the tree format, the rewrite performance decreases significantly. The performance drop is mainly due to much fewer rewrite suggestions, where the LLM no longer considers some queries as 'need to rewrite and improve'. In the extreme case where we only include a tree of cardinality values, none of the input queries are rewritten.
(3) The tuning of the LLM prompt is sensitive and have no obvious pattern. For example, 'Full_schema+Value' can achieve a comparable performance as LLM-R^2, but using the filtered schema instead results in a much lower performance. This observation is in favor of the findings in [30], that LLMs are sensitive to their prompts.

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^2** 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^2** outperforms by modifying the rule sequence and incorporating an additional "FILTER_INTO_JOIN" 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 11, it is evident that **LLM-R^2** 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^2**'s enhanced flexibility and robustness, showcasing its capability to generate more varied and effective rewrite plans.

7 CONCLUSION

In this paper, 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. The current limitation is that our LLM-R^2 exhibits higher rewrite latency compared to DB-only methods due to the time consumed by calling LLM APIs and selecting demonstrations. However, our experimental results show that this increased latency is offset by the larger reduction in execution time achieved by LLM-R^2. This demonstrates the potential of LLMs in database applications, leveraging their strong generalization and reasoning capabilities. Future improvements could include efficient demonstration selection algorithms like Faiss[14] or fine-tuning an LLM specifically for query rewriting with more datasets.

ACKNOWLEDGMENTS

This research is supported, in part, by Alibaba Group through Alibaba Innovative Research (AIR) Program and Alibaba-NTU Singapore Joint Research Institute (JRI), and the Ministry of Education, Singapore, under its Academic Research Fund (Tier 2 Awards MOE-T2EP20221-0015 and MOE-T2EP20223-0004).

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.

[6] Arian Askari, Christian Poelitz, and Xinye Tang. 2024. MAGIC: Generating Self-Correction Guideline for In-Context Text-to-SQL. *CoRR* abs/2406.12692 (2024).

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

[8] 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.

[9] Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum learning. In *ICML*, Vol. 382. 41–48.

[10] 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. In *NeurIPS*.

[11] Paola Cascante-Bonilla, Fuwen Tan, Yanjun Qi, and Vicente Ordonez. 2021. Curriculum Labeling: Revisiting Pseudo-Labeling for Semi-Supervised Learning. In *AAAI*. 6912–6920.

[12] Daniel M. Cer, Mona T. Diab, Eneko Agirre, Iñigo Lopez-Gazpio, and Lucia Specia. [n.d.]. SemEval-2017 Task 1: Semantic Textual Similarity Multilingual and Crosslingual Focused Evaluation. In *ACL*. 1–14.

[13] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek R. Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. *Proc. VLDB Endow.* 14, 13 (2021), 3376–3388.

[14] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. *CoRR* abs/2401.08281 (2024).

[15] Tianyu Gao, Xingcheng Yao, and Danqi Chen. 2021. SimCSE: Simple Contrastive Learning of Sentence Embeddings. In *EMNLP*. 6894–6910.

[16] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[17] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *SIGMOD*. 160–172.

[18] Goetz Graefe and William J. McKenna. 1993. The Volcano Optimizer Generator: Extensibility and Efficient Search. In *ICDE*. 209–218.

[19] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An Autonomous Materialized View Management System with Deep Reinforcement Learning. In *ICDE*. 2159–2164.

[20] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Yejin Bang, Andrea Madotto, and Pascale Fung. 2023. Survey of Hallucination in Natural Language Generation. *ACM Comput. Surv.* 55, 12 (2023), 248:1–248:38.

[21] 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.

[22] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *Proc. VLDB Endow.* 12, 12 (2019), 2263–2272.

[23] Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin Chen-Chuan Chang, Fei Huang, Reynold Cheng, and Yongbin Li. 2023. Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs. In *NeurIPS*.

[24] 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. In *ACL*. 4644–4668.

[25] 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 *ACL*. 142–150.

[26] Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, S. Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos A. Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. 2024. Granite Code Models: A Family of Open Foundation Models for Code Intelligence. *CoRR* abs/2405.04324 (2024).

[27] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *SIGMOD*. 39–48.

[28] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *EMNLP*. 3980–3990.

[29] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language Models Can Teach Themselves to Use Tools. In *NeurIPS*.

[30] Melanie Sclar, Yejin Choi, Yulia Tsvetkov, and Alane Suhr. 2023. Quantifying Language Models' Sensitivity to Spurious Features in Prompt Design or: How I learned to start worrying about prompt formatting. *CoRR* abs/2310.11324 (2023).

[31] Ruoxi Sun, Sercan Ö. Arik, Hootan Nakhost, Hanjun Dai, Rajarishi Sinha, Pengcheng Yin, and Tomas Pfister. 2023. SQL-PaLM: Improved Large Language Model Adaptation for Text-to-SQL. *CoRR* abs/2306.00739 (2023).

[32] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurélien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. *CoRR* abs/2302.13971 (2023).

[33] Can Wang, Sheng Jin, Yingda Guan, Wentao Liu, Chen Qian, Ping Luo, and Wanli Ouyang. 2022. Pseudo-Labeled Auto-Curriculum Learning for Semi-Supervised Keypoint Localization. In *ICLR*.

[34] 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*. ACM, 94–107.

[35] Jerry W. 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. *CoRR* abs/2303.03846 (2023).

[36] Wentao Wu, Philip A. Bernstein, Alex Raizman, and Christina Pavlopoulou. 2022. Factor Windows: Cost-based Query Rewriting for Optimizing Correlated Window Aggregates. In *ICDE*. 2722–2734.

[37] Siqiao Xue, Caigao Jiang, Wenhui Shi, Fangyin Cheng, Keting Chen, Hongjun Yang, Zhiping Zhang, Jianshan He, Hongyang Zhang, Ganglin Wei, Wang Zhao, Fan Zhou, Danrui Qi, Hong Yi, Shaodong Liu, and Faqiang Chen. 2023. DB-GPT: Empowering Database Interactions with Private Large Language Models. *CoRR* abs/2312.17449 (2023).

[38] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *ICLR*.

[39] Seonghyeon Ye, Jiseon Kim, and Alice Oh. 2021. Efficient Contrastive Learning via Novel Data Augmentation and Curriculum Learning. In *EMNLP*. 1832–1838.

[40] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic View Generation with Deep Learning and Reinforcement Learning. In *ICDE*. 1501–1512.

[41] 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. *CoRR* abs/2309.01219 (2023).

[42] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670.

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

[44] 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), 823–835.

[45] 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 (2021), 46–58.

[46] Yuhang Zhou, He Yu, Siyu Tian, Dan Chen, Liuzhi Zhou, Xinlin Yu, Chuanjun Ji, Sen Liu, Guangnan Ye, and Hongfeng Chai. 2023. R^3-NL2GQL: A Hybrid Models Approach for for Accuracy Enhancing and Hallucinations Mitigation. *CoRR* abs/2311.01862 (2023).