A Learned Query Rewrite System

Xuanhe Zhou
Tsinghua University
zhouxuan19@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University
liguoliang@tsinghua.edu.cn

Jianming Wu
Tsinghua University
wjm@yeah.net

Jiesi Liu
Tsinghua University
poldman@yeah.net

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

Xinning Zhang
Tsinghua University
zhang-xn22@mails.tsinghua.edu.cn

ABSTRACT

Query rewriting is a challenging task that transforms a SQL query to improve its performance while maintaining its result set. However, it is difficult to rewrite SQL queries, which often involve complex logical structures, and there are numerous candidate rewrite strategies for such queries, making it an NP-hard problem. Existing databases or query optimization engines adopt heuristics to rewrite queries, but these approaches may not be able to judiciously and adaptively apply the rewrite rules and may cause significant performance regression in some cases (e.g., correlated subqueries may not be eliminated). To address these limitations, we introduce `LearnedRewrite`, a query rewrite system that combines traditional and learned algorithms (i.e., Monte Carlo tree search + hybrid estimator) to rewrite queries. We have implemented the system in Calcite, and experimental results demonstrate `LearnedRewrite` achieves superior performance on three real datasets.

PVLDB Reference Format:
Xuanhe Zhou, Guoliang Li, Jianming Wu, Jiesi Liu, Zhaoyan Sun, and Xinning Zhang. A Learned Query Rewrite System. PVLDB, 16(12): 4110 - 4113, 2023.
doi:10.14778/3611540.3611633

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/zhouxh19/LearnedRewrite.

1 INTRODUCTION

Query rewriting is a critical problem in query optimization (e.g., PostgreSQL [6], Calcite [4, 8], and Soar [3]). The goal of query rewriting is to transform a SQL query at the logical level (e.g., removing redundant operators, pulling up subqueries), such that the rewritten query is equivalent to the original one and has improved execution time. However, query rewriting is an NP-hard problem [6, 8, 10, 12], and existing methods rewrite SQL queries using predefined rule orders, such as attempting to pull up the subquery before pushing down predicates. This approach is limited because it applies rewrite rules in a default order, which may result in a local optimum. For instance, as shown in Figure 1, for

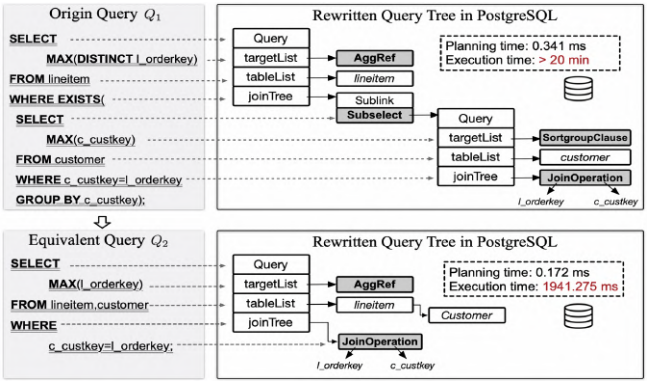

Figure 1: Example query rewrite (removing redundant subquery) with over 600x speedup before running in Postgres.

the original query Q_1, PostgreSQL rewrites Q_1 in a top-down manner, achieving only limited execution time reduction because the *MAX(l_orderkey)* operator in the subquery is not removed, and the subquery cannot be eliminated. In contrast, `LearnedRewrite` rewrites Q_1 into Q_2 by judiciously applying the rewrite rules, first removing the aggregate operator in the subquery and then merging the subquery with the join operator. For the rewritten query Q_2, `LearnedRewrite` executes Q_2 on PostgreSQL, achieving over 600x execution time reduction.

Challenges. Query rewriting for different datasets and queries can be challenging in practice. In this demonstration, we aim to solve three key challenges. Firstly, the search space of possible rewrite strategies for any query can be exponential to the number of applicable rules. Thus, a major challenge is *how to represent such a large amount of rewrites* **(C1)**. Secondly, given a large search space, another challenge is *how to find the near-optimal rewrite strategy with performance guarantees* **(C2)**. Thirdly, to select a good rewrite strategy, it is vital to estimate the benefit, or cost reduction, of a rewrite (or a sequence of rewrites), which is relevant to the query operators, rewrite rules, and dataset statistics. However, there are scenarios where we cannot access the databases or the whole dataset due to privacy issues. Thus, the third challenge is *how to estimate the cost reduction of a rewrite outside databases* **(C3)**.

Our Methodology. To tackle these challenges, we propose `LearnedRewrite`, a learned query rewrite tool that takes a SQL query and corresponding statistics (e.g., schema, #-table rows) as input, and outputs an optimized rewritten query by finding the optimal rewrite sequence. To represent the exponential number of

Proceedings of the VLDB Endowment, Vol. 16, No. 12 ISSN 2150-8097.
doi:10.14778/3611540.3611633

[1]Guoliang Li is the corresponding author.

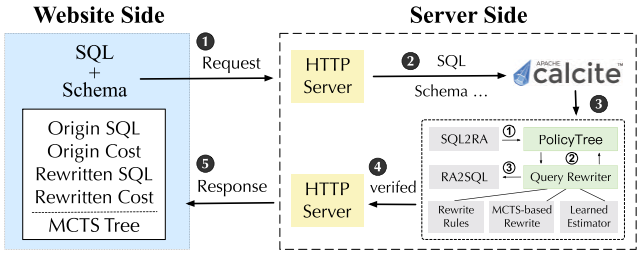

Figure 2: The Architecture of `LearnedRewrite`.

rewrite strategies, `LearnedRewrite` models the possible rewrite sequence as a policy tree, where each node corresponds to a rewritten query obtained by applying rewrite rules on its parent (addressing **C1**). We utilize Monte Carlo Tree Search (MCTS) to iteratively explore the policy tree and find the optimal node (i.e., rewritten query with maximum cost reduction) by defining node utility that considers both cost reduction and access frequency, guiding the search to find the promising rewrite sequences (addressing **C2**). To estimate the cost reduction of each node accurately, we propose a deep estimation model that considers query operators, applicable rewrites, and dataset features (addressing **C3**). In our demonstration, users can compare the SQL queries before and after rewriting and visualize the explored policy trees for different queries to better understand the rewrite results and how `LearnedRewrite` finds high-benefit rewrite strategies.

This demo is an extension of our research paper [13]. As mentioned above, `LearnedRewrite` distinguishes itself from other query rewrite tools in three main aspects: (1) `LearnedRewrite` uses a policy tree model to represent candidate rewrite sequences efficiently; (2) `LearnedRewrite` carefully applies rewrite rules to different queries and effectively improves the performance; (3) `LearnedRewrite` is an auxiliary tool implemented in Calcite, making it easy to use for various queries, datasets, and databases. Our experiments on real datasets have demonstrated that `LearnedRewrite` can efficiently rewrite queries with much higher performance than existing rewrite methods (e.g., over 25.3% latency reduction) with acceptable rewrite overhead (e.g., milliseconds).

2 LEARNEDREWRITE OVERVIEW

The architecture of `LearnedRewrite` is presented in Figure 2. Firstly, `LearnedRewrite` supports multiple users to submit their rewrite requests without accessing the databases. Secondly, `LearnedRewrite` initializes a policy tree for each submitted query and rewrites these queries with hybrid rewrite algorithms (e.g., heuristic, learned). Thirdly, `LearnedRewrite` demonstrates the rewrite results to help users easily trace the rewrite procedure.

Data Collection. To provide a lightweight query rewrite tool, `LearnedRewrite` only requires necessary statistical information from the users, such as the original queries, table schema, and row counts. Additionally, we pass the original queries into the Calcite engine and parse them into relational algebra expressions (RAs) based on the table schema. We directly apply rewrite rules on RAs instead of SQLs, as there can be information loss when translating between SQLs (e.g., removed *groupby* clauses, predicate pushdown on intermediate tables).

Query Rewrite. Next, we provide a suite of different rewrite algorithms to optimize the overall rewrite performance. For any parsed query (RA), ① if the matched rules of this query have rare overlap (i.e., each rule rewrites different operators of the query), we adopt a heuristic algorithm to apply any rewrite rules that can transform this query. ② Otherwise, we build a policy tree for this query, where the root node denotes the original query and any non-root node denotes a rewritten query that applies a rule on its parent node. However, this policy tree can be extremely large (e.g., around 10! branches with 10 applicable rules), and enumerating all possible rewrite rule sequences is not practical, especially for SQL queries with dozens of operators. To efficiently obtain the optimal node in the policy tree, we propose a *Monte Carlo Tree Search* (MCTS) based search strategy that judiciously explores the nodes to obtain the optimal node. MCTS [9] is a well-known tree search algorithm that balances exploitation (high benefit) and exploration (low frequency) when searching the tree and can acquire more information about the "optimal" node, i.e., the optimal strategy of applying available rewrite rules. Moreover, to provide effective guidance on the explored rewrites, we propose a self-attention-based estimator that captures the relations between complex queries and rules and can provide relatively accurate benefit estimation (i.e., cost reduction caused by the rewrites).

Web Service. We provide a web interface to help users easily and interactively check out the query rewrite procedure. In the web interface, we provide three main functions: (*i*) Submit query rewrite requests; (*ii*) Formalize submitted query (e.g., removing redundant whitespaces); (*iii*) Demonstrate the rewritten query together with the reduced costs; (*iv*) Demonstrate the policy tree of the original query, which exposes the explored rewrite sequences and why `LearnedRewrite` has adopted the final rewritten query.

3 DEMONSTRATION

In this section, we showcase the user interface of `LearnedRewrite`, where users can submit their queries and observe the rewrite procedure. The online website will be continuously updated.

3.1 End-to-End Experience.

Figure 3 is a screenshot of the front-end of `LearnedRewrite`. The user can pose rewrite requests of different queries and schema and observe the rewrite procedure with the following steps.

1) Customize Table Schema. Before submitting origin queries, users need to define the schema, which is passed as the planner configuration. If the schema information is not available due to privacy issues, we also provide a testing schema (extracted from TPC-H), based on which users can modify their queries on TPC-H tables and run the modified queries on `LearnedRewrite` (Fig. 3-①).

2) Input And Transform Origin Query. Next users input their queries. Here, we provide three transformation functions (Fig. 3-②): (*i*) *Query Formatting*. Since user-crafted queries can be very casual and confusing (e.g., no line breaks, random whitespaces), users can click the “format” button, and `LearnedRewrite` will automatically format the query statement (e.g., adding proper line breaks) [1]; (*ii*) *Query Parsing*. Since query rewrite works on relational algebra expressions, many rewrites cannot be directly reflected by SQL statements (e.g., some queries before/after pushing down predicates corresponding to the same SQL statement). Thus, users can click the “parse” button and check out the RA format query (*RelNode* in Calcite); (*iii*) *Query Rewriting*. After parsing the query into RA

format, users can click the "rewrite" button, and LearnedRewrite will start transforming the RA query in logical level.

3) Check Recommended Rewriting. After the *Query Rewriting* procedure finishes, users can check out the rewritten query recommended by LearnedRewrite. In this page, LearnedRewrite demonstrates two comparisons (Fig. 3-③): (*i*) The reduced costs before/after rewriting, which are approximated by our learned estimator. Note the estimator guides the rewriting directions and so can significantly affect the rewriting results. Thus, we also provide the "Cost Function" button, with which users can replace into similar cost functions of their databases or learned model file; (*ii*) The SQL statements before/after rewriting, which users can easily execute on their databases; (*iii*) The parsed queries before/after rewriting, whose differences are marked with different colors by LearnedRewrite. The parsed queries can supplement changes that cannot be reflected by the SQL statements. Besides, we also provide the description of the rewriting results in natural languages, which is supported by the large language model [7] (Fig. 3-④).

4) Check Rewriting Procedure. Besides the final rewritten queries recommended by LearnedRewrite, users may be interested in the complete rewriting procedure. Thus, users can click the "report" button and LearnedRewrite will demonstrate the policy tree of the target query (a rule path for heuristic rewriting), from which users can observe the explored rewrite sequences together with their rewritten queries and reduced costs. This may be useful to identify valuable rewrite rule combinations for typical query patterns. Additionally, the report also provides a visualization of the policy tree to show the rewrite procedure step by step (Fig. 3-⑤).

3.2 Scenario 1 - Testing Queries on Postgres.

Standard TPC-H benchmark contains 22 template queries, which involve relatively complex structures (e.g., multi-join+aggregates in Q_5, multi-join+subqueries in Q_2) and are suitable to conduct query rewrite. However, most database vendors polish their rewriters (optimizers) based on the 22 TPC-H queries and we need more queries to verify the rewrite capacity in general scenarios. Thus, we adopt two query generation approaches in LearnedRewrite:

1) Random Testing Queries. Intuitively we can utilize SQLsmith to randomly generate complex queries. However, these queries may have many redundant operators, e.g., the filter operators output zero tuples and the left operators are not executed. By removing those redundant operators (rewriting benefits by estimator or running on sample data), LearnedRewrite can target at more beneficial operators (e.g., the filters) and effectively rewrite these operators (e.g., extracting atomic predicates);

2) Testing Queries with Typical Structures and Cost Ranges. We also support reinforcement learning algorithm that generates queries with both various structures and different cost ranges [11]. Compared with SQLsmith, users can easily generate desired SQL queries by customizing (*i*) the SQL syntax and (*ii*) cost/cardinality constraints based on their scenario characters, and optimize these SQL queries with LearnedRewrite.

3.3 Scenario 2 - Real OLTP Queries on Oracle.

Compared to testing queries, rewriting real-world queries can be more challenging. Firstly, queries from multiple applications may have complex semantics that are difficult to capture from SQL statements. LearnedRewrite can help to judiciously rewrite these queries and highlight the modified parts to enhance user understanding. Secondly, real queries often use placeholders for values, which are replaced before they become publicly available. LearnedRewrite needs to analyze the statistics (e.g., value ranges of the columns, #-accessed-blocks), infer the removed values, and then rewrite the queries. In these cases, the policy tree can assist users in selecting the most appropriate rewritten queries.

Future Revolution. Furthermore, there are still many real-world requirements for query rewriting. For instance, more complex SQL grammars like UDFs involve relatively complex logic (e.g., iterations), and existing rewrite rules cannot handle them. Thus, it is necessary to generate rewrite rules (or provide instructions) for these complex SQL queries. Additionally, many engineers use object-oriented paradigms (e.g., ORM) to write database requests. In the future, LearnedRewrite will integrate more functions (e.g., SQL2ORM) to support more general database applications.

4 EVALUATION

Settings. We evaluate the rewriting capability of LearnedRewrite in comparison with Postgres' default rewriter, which conducts logical transformation rules over parse trees in two stages before query planning. We use 15 queries from real datasets, i.e., *Shopmall* (78 tables with 298,208 sampled tuples) and *Material* (41 tables with 683,130 sampled tuples). The evaluation is performed on a machine with 16GB RAM, 256GB disk, 4.00GHz CPU.

SQL Verification. Some rules in Calcite may not ensure semantic equivalence (e.g., the result set changes), we try to find and remove those rules by utilizing some basic proof assistants to reason about the conducted rewrites (e.g., operators reordering, operator elimination) [2, 5]. And since SQL verification is a complex issue, in some cases we still require users to verify by actually running on their databases (e.g., significant changes occur in the rewritten queries).

Performance Analysis. Figure 4 shows the detailed latency reduction of queries in the real datasets. And we find most of the queries (80%) can be optimized with LearnedRewrite. And we showcase some typical rewrites together with the insights.

Query#1 aims to count the number of products that satisfy certain conditions in a specific shop. The origin query uses inner joins to combine data from three tables (i.e., shop_goods, mall_goods, and mall_picture), and filter the results based on the given conditions. Instead, LearnedRewrite rewrites the join between the shop_goods, mall_goods, and mall_picture tables using a subquery, i.e., creating a subquery that filters the shop_goods table based on the conditions in the WHERE clause, and then join that result with the mall_goods table. This way, LearnedRewrite uses subqueries and joins to reduce the number of tables that need to be scanned and simplify the conditional statement.

Query#6 aims to select unique non-null supplier values from two tables, combining them using a union, and assigning alias names to the resulting supplier column. The rewritten query eliminates a DISTINCT operator by using UNION ALL instead of UNION, and also removes the GROUP BY clause.

Query#12 aims to select the count of rows that satisfy some conditions across four tables. The rewritten query is simpler and much faster because it avoids unnecessary join operations and

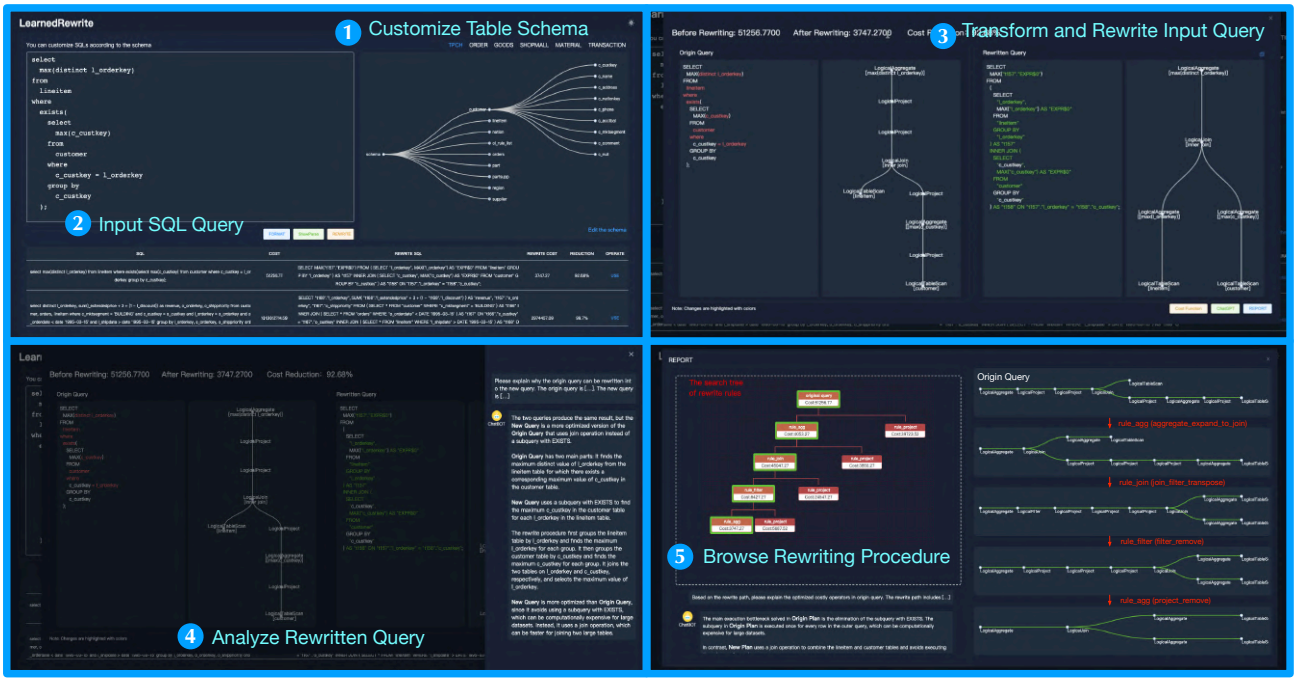

Figure 3: A Screenshot of LearnedRewrite (http://rewrite_demo.dbmind.cn/)

Figure 4: Performance on real datasets. `LearnedRewrite` reduces the execution latency of 12 out of the 15 slow real queries, resulting in over 25.3% total latency reduction (detailed queries in github.com/zhouxh19/LearnedRewrite/tree/main/real).

filters early by only considering rows that have a matching value in one of the four tables. The rewritten query returns a count of 0 for the specified conditions because the fake table has no rows. The difference in latency between the original and rewritten queries is significant because the rewritten query requires much less computation and processing.

Query#8 aims to selects specific column values from a table. *The rewritten latency is higher than the origin latency even though the rewritten query has a lower estimated cost than the original query.* In this case, the rewritten query includes an explicit order of the IN clause values whereas the original query does not. This could result in a different execution plan being chosen by the query optimizer which could have a different impact on performance. Additionally, the rewritten query might have a larger result set due to the changed ordering of the IN clause values, which could result in higher latency due to increased I/O or network traffic.

ACKNOWLEDGEMENTS

This paper was supported by NSFC(61925205, 62232009, 62102215), Huawei, TAL education, and Zhongguancun Laboratory.

REFERENCES

[1] https://github.com/jdorn/sql-formatter (last checked on 2023-7).
[2] https://github.com/uwdb/cosette (last checked on 2023-7).
[3] https://github.com/xiaomi/soar (last checked on 2023-7).
[4] E. Begoli, J. Camacho-Rodríguez, J. Hyde, and et al. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *SIGMOD*, pages 221–230, 2018.
[5] S. Chu, K. Weitz, A. Cheung, and D. Suciu. Hottsql: proving query rewrites with univalent SQL semantics. In *PLDI*, pages 510–524, 2017.
[6] B. Finance and G. Gardarin. A rule-based query rewriter in an extensible DBMS. In *ICDE*, pages 248–256. IEEE Computer Society, 1991.
[7] J. Li and et al. Can LLM already serve as A database interface? A big bench for large-scale database grounded text-to-sqls. *CoRR*, abs/2305.03111, 2023.
[8] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In M. Stonebraker, editor, *SIGMOD*, pages 39–48, 1992.
[9] I. Trummer, J. Wang, D. Maram, and et al. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In *SIGMOD*, pages 1153–1170. ACM, 2019.
[10] Z. Wang, Z. Zhou, Y. Yang, and et al. Wetune: Automatic discovery and verification of query rewrite rules. In *SIGMOD*, pages 94–107, 2022.
[11] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware SQL generation using reinforcement learning. In *SIGMOD*, pages 945–958, 2022.
[12] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *TKDE*, 34(3):1096–1116, 2022.
[13] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow.*, 15(1):46–58, 2021.