Verified LLM-Based Query Rewriting for Microsoft SQL Server

Kukjin Lee
Microsoft
kulee@microsoft.com

Anshuman Dutt
Microsoft
andut@microsoft.com

Vivek Narasayya
Microsoft
viveknar@microsoft.com

Surajit Chaudhuri
Microsoft
surajitc@microsoft.com

ABSTRACT

Query rewriting is one of the techniques used by database developers and administrators to tune the performance of poorly performing SQL queries. We have developed an LLM-based query tuning tool for Microsoft SQL Server that takes as input a query, tuning constraints such as minimum required improvement and time budget, and automatically recommends semantically equivalent query rewrites that improve the execution time of the query. The software tool is incorporated as an extension to SQL Server Management Studio (SSMS). Our demonstration highlights two important capabilities. First, while LLMs have the ability to generate promising candidate rewrites for the given query, they cannot guarantee semantic equivalence of a query rewrite. We use a technique we developed for verifying semantic equivalence of two queries, which leverages the built-in capabilities of Microsoft SQL Server's query optimizer. Second, we judiciously use the given time budget by adopting an anytime tuning algorithm and employing optimizations that help short-circuit the expensive steps of verification and execution for each candidate rewrite. To the best of our knowledge this is the first tool of its kind that enables DBAs and application developers to safely operationalize LLM-based query rewriting.

PVLDB Reference Format:
Kukjin Lee, Anshuman Dutt, Vivek Narasayya, and Surajit Chaudhuri. Verified LLM-Based Query Rewriting for Microsoft SQL Server. PVLDB, 19(12): 4594-4597, 2026.
doi:10.14778/3827998.3828074

1 INTRODUCTION

Query rewriting is widely used by database developers and administrators to improve SQL query performance when the optimizer-generated plan is suboptimal. The goal of query rewriting is to generate a semantically equivalent query with significantly lower execution time. Recently, there has been a large body of work on the use of Large Language Models (LLMs) for tuning database performance [7, 8, 18, 21, 23]. For the problem of query rewriting, recent work [5, 11, 12, 19] has developed techniques that, for a given query Q, use an LLM to generate a rewritten query Q'. A recent paper [15] found that even on real-world enterprise databases and queries, that LLMs have not been trained on, LLMs were able to generate query rewrites that execute significantly faster.

Despite the promise shown by LLMs in query rewriting, there are two major impediments in the adoption of LLM-generated query rewriting in practice. First, the semantic equivalence of the LLM-generated rewrite is not guaranteed. Furthermore, due to the complexity of SQL queries encountered in real-world applications, and the subtleties of correctly handling NULLs and empty results in SQL, it is not practical for DBAs or developers to manually verify semantic equivalence of a query rewrite. Second, users not only want verified rewrites, they also expect the rewritten query to execute (significantly) faster than the original query. Ensuring this requires potentially generating and executing multiple different LLM-generated query rewrites, which can be expensive.

We have developed a software tool that leverages the effectiveness of LLMs in generating query rewrites, but alleviates the above two impediments. Users provide as input a SQL query, constraints including the time allowed for tuning, and minimum improvement in execution time required. The tool searches through a space of rewrites and outputs: (a) query rewrites that are *verified to be semantically equivalent*, and (b) satisfying the constraints on minimum improvement required. It uses a novel mechanism for checking semantic equivalence of two queries, called QO-Verify (described in [15]). QO-Verify re-purposes the query optimizer of Microsoft SQL Server, and specifically uses the *memo* data structures generated for the original and rewritten query to check for semantic equivalence. Importantly, QO-Verify is able to check for semantic equivalence for complex queries with joins, group-by and aggregation, nested subqueries etc., which state-of-the-art SQL equivalence checkers e.g., [3, 20] fail to verify. To improve efficiency, the tool uses several techniques to avoid expensive query executions by: (i) detecting duplicate plans across rewrites (ii) set execution timeouts leveraging the current best rewrite's execution time and the desired execution time improvement specified by the user. The tool has an *anytime* behavior which means that users can stop the tool at any time during the tuning, and get the best rewrite thus far.

To the best of our knowledge, this tool is the first of its kind that can "close the loop" by safely and efficiently operationalizing LLM-based query rewriting. The tool is built as an extension to SQL Server Management Studio (SSMS), a popular GUI tool used extensively by DBAs and database developers. Our demo will highlight the effectiveness of LLM-generated rewrites, showcase the semantic equivalence, efficiency and anytime behavior of the tool and explain opportunities for future work.

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828074

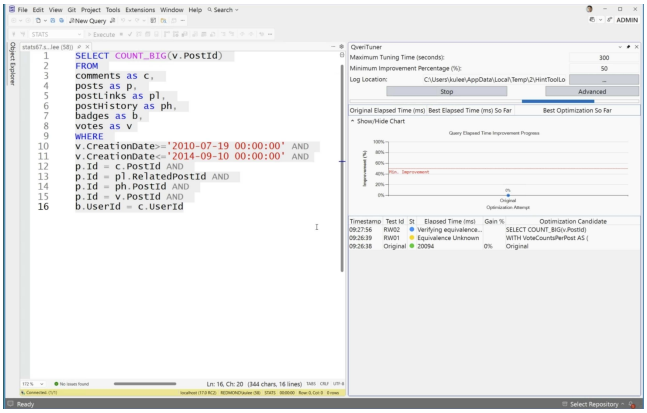

Figure 1: Original query in LHS pane, and tuning progress in RHS pane

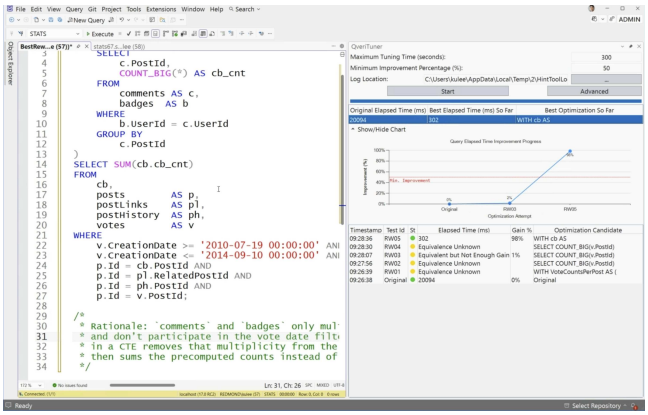

Figure 2: Best rewritten query with explanation, verification check status and elapsed time

2 FUNCTIONALITY

The tool is an interactive, anytime query tuning solution that assists database developers and administrators in improving query performance using LLM-driven query rewriting, while guaranteeing semantic equivalence. Figure 1 shows a screenshot of the tool, an extension in SQL Server Management Studio (SSMS). As shown in Figure 1, the user provides as input: (i) A SQL query whose performance they want to improve, (ii) A minimum required improvement in execution time for the rewritten query (e.g., 50% reduction in execution time), and (iii) A maximum tuning time budget (e.g. 300 seconds). These inputs define the optimization envelope within which the tool operates. The tool invokes a large language model (LLM) to generate candidate query rewrites iteratively, while ensuring that only rewrites that are both semantically equivalent and improve performance sufficiently are considered for recommendation. The input query is shown in the LHS pane, and the tuning progress is shown in the RHS pane.

The screenshot in Figure 2 illustrates the presentation of results in the right-hand side pane of the interface. The pane displays the best verified rewrite found so far, along with its measured execution time and improvement relative to the original query. In addition, the tool asks the LLM to include, as a comment in the rewrite, a brief rationale explaining why the rewritten query is expected to be faster. This rationale is intended to improve transparency and help users understand the potential performance implications of the rewrite, without requiring them to manually analyze complex execution plans. Alternative rewrites explored by the tool, along with their execution time and verification status are also shown in the RHS pane. This allows the user to choose an alternative rewrite if it is more appropriate for their application use case.

The tool 's behavior is configurable through a set of advanced options, which allow users to control the space of rewrites explored by the tool. These options specify which classes of query rewrites the LLM is permitted to consider, such restructuring a query to use CTEs, exploiting shared expression across subqueries, the introduction of query hints, etc. By constraining the rewrite space, users can tailor the tool's behavior to their workload characteristics or organizational policies, while still benefiting from automated verification and performance evaluation. An overview of the optimizations used by the tool to improve efficiency, techniques used for verifying semantic equivalence of a rewrite, as well as prompts used for LLMs are described in Section 3.

Overall, the tool tightly integrates LLM-based query rewriting with optimizer-based equivalence verification and efficiently performing empirical performance testing. By automatically closing the loop between rewrite generation, semantic verification, and execution time measurement, the tool eliminates the need for manual validation and trial-and-error tuning. This design enables DBAs and developers to safely exploit the power of LLMs for query optimization, while retaining strong correctness guarantees.

2.1 Example walkthrough

We explain the workflow using a simple query on *StackOverflow* dataset [9] with tuning budget 300 seconds and minimum expected improvement 50%. The tool first establishes baseline execution time of the original version of the query to be 20 seconds. Then it starts iterating over a pre-defined sequence of rewrite prompts to LLM. As shown in RHS, the tool skips the execution of rewritten queries that could not be verified to be equivalent to the original query and executes only two rewrites (RW03, RW05). The third prompt results in a query that is equivalent to the original but does not deliver the expected performance improvement. Finally, the fifth rewrite prompt delivers an equivalent as well as performant rewrite, improving over 98% compared to the original query. The entire process completes using 5 calls to LLM and QO-verify, and 2 plan executions, saving 3 executions due to our algorithmic improvements. The tool also outputs an rationale for the performance gains, which in this case occurs by by pushing down partial aggregate before the join size explodes due to many-to-many joins. The end-to-end workflow completed in 78 seconds across LLM calls, query executions, and equivalence verification calls (in decreasing order of magnitude).

2.2 Demonstration scenarios

We will demonstrate the functionality of the tool, including:

(1) Run the tool on various SQL queries, visualize the tuning process, and see the recommended rewritten query along with

Algorithm 1 Anytime Algorithm of the tool

Require: Original query Q_{orig}, minimum improvement percentage M, time budget $T_{\max}$
Ensure: Best verified rewrite found so far Q_{best}
1: $Q_{\text{best}} \leftarrow Q_{\text{orig}}$
2: $E_{\text{best}} \leftarrow \textsc{Execute}(Q_{\text{orig}})$ ▷ Baseline elapsed time; skip if already available
3: timeout $\leftarrow (1 - M/100) \cdot E_{\text{best}}$
4: $\mathcal{P} \leftarrow \emptyset$ ▷ Set of previously seen execution plans
5: **while** Time budget remains and rewrites are available **do**
6: $Q_{\text{rewrite}} \leftarrow \textsc{NextRewriteFromLLM}(Q_{\text{orig}})$
Skip rewrite
7: **if** $\textsc{IsHint\_NoPlanChangeExpected}(Q_{\text{rewrite}})$ **then**
8: **continue** ▷ Hint skipping optimization
9: $P_{\text{rewrite}} \leftarrow \textsc{Optimize}(Q_{\text{rewrite}})$
10: **if** $P_{\text{rewrite}} \in \mathcal{P}$ **then**
11: **continue** ▷ Duplicate plan detection
12: $\mathcal{P} \leftarrow \mathcal{P} \cup \{P_{\text{rewrite}}\}$
Verify equivalence
13: **if** $\textsc{QOVerify}(Q_{\text{orig}}, Q_{\text{rewrite}}) \neq$ **Yes then**
14: **continue** ▷ Soundness: skip unverified rewrites
Execute query
15: $E \leftarrow \textsc{ExecuteWithTimeout}(Q_{\text{rewrite}}, \text{timeout})$
16: **if** $E < E_{\text{best}}$ **then**
17: $Q_{\text{best}} \leftarrow Q_{\text{rewrite}}$
18: $E_{\text{best}} \leftarrow E$
19: timeout $\leftarrow E$ ▷ Timeout reduction
20: **return** Q_{best}

the rationale for the rewrite. We will compare the plans of the original and rewritten queries.
(2) Analyze each LLM recommended rewrite with details such as verification status and execution time, observe anytime behavior and analyze impact of overhead reduction techniques.
(3) Visualize the equivalence verification steps in QO-Verify to explain how the technology works.

3 TECHNIQUES

We provide a brief overview of the algorithm used by the tool, and we briefly discuss some of the key technical challenges addressed.

3.1 Anytime algorithm of the tool

An outline of the algorithm used by the tool is shown in Algorithm 1. The algorithm is *anytime*, i.e. can be stopped at any moment and still return the best solution it has found so far, with solution quality monotonically improving the longer it runs.

The tool first executes the original query to establish a baseline of elapsed time to execute the query $Elapsed_{QOrig}$. Note that if such a baseline already exists, e.g., as part of historical execution information in Query Store, this step can potentially be skipped. After this the program executes a loop until time runs out, or no more rewrites are available. Now we describe a few important aspects of the algorithm related to efficiency and soundness.

Timeout reduction: The input to the tool of minimum performance improvement percentage M required is with respect to $Elapsed_{QOrig}$. Therefore, in step 3 the tool initializes the timeout for future executions as $(1 - \frac{M}{100})Elapsed_{QOrig}$. Thus, no rewrite explored by the tool ever needs to execute longer than this timeout value. Note that if any rewrite is found to execute faster than the current timeout value, then the timeout is further reduced for subsequent executions (see Step 19). In practice, once even a single rewrite is found that is significantly faster, it greatly speeds up the execution of the rest of the algorithm.

Hint skipping: We developed one optimization that applies to use of query hints only that is effective in practice. If the plan for Q_{orig} cannot be altered by the application of a specific hint (or hint combination), then we can skip it. As a simple example, suppose the hint is to force the physical operator to be Hash Join only, but the current plan for Q_{orig} already uses Hash Join only, then the hint can be skipped.

Duplicate plan detection: Once a rewrite $Q_{rewrite}$ is found, if the execution plan for $Q_{rewrite}$ is identical to the execution plan of an earlier rewrite already explored, we skip such rewrites immediately. This optimization fires quite often in practice, since different rewrites or query hints can result in the same execution plan.

Soundness of rewrite: While the algorithm explores different query rewrites using the LLM, we cannot be sure if rewrite proposed by the LLM $Q_{rewrite}$ is semantically equivalent to Q_{orig}. In practice DBAs or application developers cannot use a query rewrite unless it is proven to be semantically equivalent. Although there has been prior work to verify semantic equivalence of two queries, e.g., [3, 20], as noted in [5, 15] in practice their coverage does not extend to queries containing aggregation, outer joins, nested subqueries etc. Therefore, for the purposes of verifying semantic equivalence we use the QO-Verify technique [15]. This mechanism either returns Yes if it can prove the two queries to be semantically equivalent or Unknown otherwise. As part of this work, we integrated QO-Verify into the SSMS extension as a dynamically linked library (DLL). For completeness, we include a brief overview of QO-Verify below.

3.2 Verifying semantic equivalence

QO-Verify [15] checks semantic equivalence of two SQL queries by reusing the query optimizer's memo structure, the internal data structure that compactly represents large spaces of logically equivalent query expressions explored during optimization. Given an original query Q and a candidate rewrite Q', QO-Verify invokes the optimizer to build a memo for each query. Intuitively, QO-Verify checks if there is at least one logical expression in common among all the logical expressions enumerated by the optimizer for Q and Q'. Since each transformation rule is known to be semantics preserving, finding a common logical expression implies that the two queries are semantically equivalent. Because the memo already encodes the optimizer's full understanding of SQL semantics—including NULL handling, outer joins, aggregation, and correlated subqueries, QO-Verify provides a sound equivalence test through a radically different approach compared to prior work that uses SMT solvers, symbolic reasoning or testing.

A key strength of QO-Verify is that it reuses decades of engineering investment in developing transformation rules in industrial query optimizers, thereby allowing it to verify complex, real-world queries. In [15], QO-Verify is shown to be effective at validating LLM-generated query rewrites on several real-world queries on Microsoft SQL Server, thereby removing a major barrier to adopting LLM-based query rewriting in practice. The median time to verify semantic equivalence is reported to be around 0.85 seconds, which is acceptable for offline query tuning scenarios such as in the tool.

3.3 LLM prompts and models

The tool is configured with several different prompts, which is based upon prompts described in [5, 12]. (1) General (2) CTE (3) SubQ. We add a new prompt: (4) QHint. All prompts request the LLM to produce a semantically equivalent rewrite whose performance is significantly better than the original query. The prompt also requests the LLM to return a rationale explaining why it recommended the rewrite. General is unconstrained, i.e., it provides no further guidance or constraints. CTE requests creation of Common Table Expression(s) to express repeated computations within the original query. We observe that unlike in PostgreSQL, Microsoft SQL Server does not automatically materialize CTEs specified in the query, but uses the Spool operator in a cost-based manner. SubQ specifically targets optimizing multiple shared or similar expressions in subqueries. QHint requests the LLM to recommend zero or more query hints from the set available in Microsoft SQL Server [13]. We augment each prompt with additional information: (a) The total number of rows of each table referenced in the query. (b) Information extracted from executing the plan for the original query specifically: the logical/physical operators in the plan, and for each operator its estimated cost, estimated cardinality and actual cardinality. We also have configured the tool so that users can choose from a variety of models, and we use GPT-5.2 from OpenAI as the default [16, 17].

4 RELATED WORK

LLM-based query rewriting: GenRewrite[12] creates Natural Language Rewrite Rules (NLR2s) which are passed to the LLM in the prompt. They conduct multiple iterations of prompting to improve the quality of the rewritten query. LITHE [5] uses different prompts that request the LLM to remove redundancy within a query. They also develop an MCTS-based technique for generating token probabilities to guide LLM inference to improve accuracy of rewrite. Our work leverages ideas for prompts developed by the above prior work, which we augment with plan specific information, as well as new prompts for query hinting. LLMSTEER[1] uses LLMs for query hinting. However, unlike our approach which uses the LLM to generate query hint recommendations, they use the LLM to create an embedding vector of the query, and train an ML model using the embedding that predicts which query hint to use.

Query Hint recommendation tool: The Query Hint Recommendation tool (QHRT) [14] for Microsoft SQL Server recommends appropriate query hints for a query. In contrast, the tool significantly broadens the functionality and scope of QHRT by incorporating: (a) query rewriting using LLMs and (b) semantic verification.

Equivalence of SQL queries: Several techniques have been developed for proving equivalence of two SQL queries e.g., UDP[2], SPES[22], Cosette[3], QED[20], SQLSolver[6], and VeriEQL[10]. One common approach is to convert each query into a logic formula, and use SMT solvers (e.g., [4]) to check if the formulas are equivalent. However, these solvers are limited in the class of queries that can be verified automatically, and particularly fall short for queries with aggregation. As noted in [5, 15], these methods fail on all TPC-DS and real-world query rewrites.

Using LLMs for query tuning: There is a significant body of work on using language models for performance tuning and query optimization in databases. Some examples include: Panda[18], which uses LLMs for performance debugging, DBG-PT[7] for diagnosing performance regressions, $\lambda - Tune$ [8] for tuning database knobs, DB-GPT[23] for index tuning. An alternative approach to using LLMs off-the-shelf is proposed in [21] to train foundation models to learn representations of data, logical and physical query plans for use in multiple tuning tasks. These are complementary to query tuning using LLM-based rewriting.

REFERENCES

[1] Peter Akioyamen, Zixuan Yi, and Ryan Marcus. 2024. The unreasonable effectiveness of llms for query optimization. *arXiv preprint arXiv:2411.02862* (2024).

[2] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *arXiv preprint arXiv:1802.02229* (2018).

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

[4] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[5] Sriram Dharwada, Himanshu Devrani, Jayant Haritsa, and Harish Doraiswamy. 2026. LITHE: A Query Rewrite Advisor using LLMs. *EDBT 2026* 29, 2 (2026), 233–246.

[6] Haoran Ding, Zhaoguo Wang, Yicun Yang, Dexin Zhang, Zhenglin Xu, Haibo Chen, Ruzica Piskac, and Jinyang Li. 2023. Proving query equivalence using linear integer arithmetic. *PACMOD* 1, 4 (2023), 1–26.

[7] Victor Giannakouris and Immanuel Trummer. 2024. Dbg-pt: A large language model assisted query performance regression debugger. *Proceedings of the VLDB Endowment* 17, 12 (2024), 4337–4340.

[8] Victor Giannakouris and Immanuel Trummer. 2025. λ-tune: Harnessing large language models for automated database system tuning. *Proceedings of the ACM on Management of Data* 3, 1 (2025), 1–26.

[9] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. (2021).

[10] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. [n.d.]. VeriEQL: Bounded equivalence verification for complex SQL queries with integrity constraints. *Proceedings of the ACM on Programming Languages* 8, OOPSLA1 ([n. d.]).

[11] Zhaodonghui Li, Haitao Yuan, Huiming Wang, Gao Cong, and Lidong Bing. 2024. LLM-R2: A Large Language Model Enhanced Rule-based Rewrite System for Boosting Query Efficiency. CoRR abs/2404.12872 (2024).

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

[13] Microsoft. 2023. Hints (Transact-SQL) - Query. https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16 Last accessed: July 8, 2026.

[14] Microsoft. 2025. Query Hint Recommendation Tool. https://learn.microsoft.com/en-us/ssms/query-hint-tool/hint-tool-overview Last accessed: July 8, 2026.

[15] Vivek Narasayya and Surajit Chaudhuri. 2026. Leveraging Query Optimizers to Verify the Soundness of LLM-based Query Rewrites for Real-World Workloads, and More!. In *16th Conference on Innovative Data Systems Research, CIDR*. 18–21.

[16] OpenAI. 2025. GPT-4o-mini. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence Last accessed: July 8, 2026.

[17] OpenAI. 2025. GPT-5. https://openai.com/index/introducing-gpt-5 Last accessed: July 8, 2026.

[18] Vikramank Singh, Kapil Eknath Vaidya, Vinayshekhar Bannihatti Kumar, Sopan Khosla, Murali Narayanaswamy, Rashmi Gangadharaiah, and Tim Kraska. 2024. Panda: Performance debugging for databases using LLM agents. (2024).

[19] Zhaoyan Sun, Xuanhe Zhou, Guoliang Li, Xiang Yu, Jianhua Feng, and Yong Zhang. 2024. R-bot: An llm-based query rewrite system. *arXiv preprint arXiv:2412.01661* (2024).

[20] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A powerful query equivalence decider for SQL. *VLDB* 17, 11 (2024), 3602–3614.

[21] Johannes Wehrstein, Carsten Binnig, Fatma Özcan, Shobha Vasudevan, Yu Gan, and Yawen Wang. 2025. Towards Foundation Database Models. CIDR.

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

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