Leveraging Query Optimizers to Verify the Soundness of LLM-based Query Rewrites for Real-World Workloads, and More!

Vivek Narasayya
Microsoft Corporation

Surajit Chaudhuri
Microsoft Corporation

ABSTRACT

Query rewriting is one of the techniques used by application developers and DBAs to tune poorly performing queries. Recently, LLM-based query rewriting techniques have been proposed, and these show significant performance improvements on industry benchmark queries. We conduct an extensive empirical evaluation on Microsoft SQL Server using real-world queries on private enterprise databases in order to quantify the effectiveness of LLM-based query rewriting. We find that LLM-based rewriting shows promise even in real-world queries. However, since LLMs cannot guarantee semantic equivalence of the rewrite, checking if the rewritten query is indeed equivalent is a major impediment that limits the practical adoption of LLM-based rewriting today. We present a sound and efficient technique that leverages built-in capabilities of the query optimizer to verify semantic equivalence. Finally, we observe that LLM-based rewrites can be a source from which query optimizer developers can identify candidate transformation rules which are missing from their optimizer. We present a few such rules we identified based on our experiences with LLM-based rewrites for real-world and benchmark queries in Microsoft SQL Server.

CCS CONCEPTS

• **Information systems → Query optimization**.

KEYWORDS

Query optimizer, transformation rules, query rewriting, large language models, AI

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 whose execution cost is significantly lower. Recently, there has been a large body of work on the use of Large Language Models (LLMs) for tuning database performance [18, 19, 37, 41, 45]. For the problem of query rewriting, recent work [12, 27, 28, 38] has developed techniques that, for a given query Q, use an LLM to generate a rewritten query Q'. These papers evaluated their techniques primarily using queries from industry benchmarks such as TPC-H, TPC-DS [31], DSB [13] and JOB [26] using PostgreSQL. For example, GenRewrite [28] finds on PostgreSQL that 21 TPC-DS queries (out of 99) achieved a speedup of at least 2× by rewriting, and the LITHE approach [12] reports on PostgreSQL that 26 TPC-DS queries achieved a speedup of at least 1.5× by using LLM-generated rewrites.

Focusing on industry benchmarks and open source databases is valuable since the results can be reproduced by the database community. However, the schema and queries of industry benchmarks are publicly available, and articles, blogs describing how to tune these queries have most likely been used for training LLMs. In contrast, enterprise data and queries are not available in the public domain. Except for the evaluation on a single proprietary customer database on a commercial DBMS [12], we are not aware of any other studies on the effectiveness of LLM-based rewriting on enterprise workload "out of the box". In this paper we present a broader empirical study across multiple real-world workloads, which LLMs have not been trained on. We conduct experiments on 118 real-world queries over four different real-world databases in addition to 224 queries on JOB, TPC-DS and TPC-H benchmarks. The real-world queries contain operations such as inner and outer joins, group-by and aggregation, and use nested sub-queries, views and common table expressions (CTEs). We ran our experiments on a commercial database system, Microsoft SQL Server. The detailed setup and results of this evaluation are described in Section 4. We were pleasantly surprised to observe meaningful improvements from LLM-based rewriting "out-of-the-box" for several real-world queries across all databases. However, we also observe that a large fraction of the rewrites (≈32%) produce different results compared to the original query on the given database, and are therefore incorrect. Even for queries that benefit from rewriting, multiple invocations to the LLM (with adjustments in prompts) are often needed to obtain a rewrite with significantly better performance. Note that the expensive step of executing the queries generated by the LLM on the database is required to be confident of the value of the LLM-based rewriting. Our empirical study examines multiple facets, e.g., distribution of improvements from rewriting of real vs. benchmark queries, distribution of improvements across databases, how restricting the scope of LLM based rewriting through prompts impacts performance, and the impact of the choice of LLM used (GPT-4o-mini vs. GPT-5 [32, 33]).

While using LLMs for query rewriting has shown promise, a major impediment to using such LLM-based query rewriting techniques in practice is the difficulty of verifying semantic equivalence of the original and rewritten query. Ensuring that the original and the rewritten query result in the same answer set over the given database (or a set of test databases) is only a necessary condition but not sufficient for equivalence. Application developers or DBAs may be able to manually verify equivalence; however, such manual verification is challenging and error-prone, particularly when the query is large or complex. Such manual verification could be aided by automated *testing*, e.g., [8] that generate counter-examples, but

 16th Annual Conference on Innovative Data Systems Research (CIDR '26). January 18-21, Chaminade, USA

such tools cannot prove equivalence. Furthermore, many applications or services may need to tune queries automatically in a closed loop with no human intervention. In all such cases, it is imperative to *automatically verify* that the rewritten queries are equivalent. The general problem of verifying if two SQL queries are equivalent is undecidable. There has been a long line of work in developing tools that use SMT solvers to check for query equivalence of two queries, e.g., [10, 15, 22, 39]. However, as noted in [12], and confirmed by our experiments, these techniques have limited coverage on industry benchmarks and real-world queries that contain group-by, aggregation, outer-join, or nested subqueries. For example, on TPC-DS the state-of-the-art query equivalence checkers are unable to verify equivalence for any of the LLM generated rewrites.

In this paper, we propose a technique to verify equivalence of queries by leveraging *built-in capabilities* of today's query optimizers. We believe this to be an overlooked opportunity which takes advantage of the extensive set of sophisticated algebraic *transformation rules* that already exist in optimizers, which were developed over the years to optimize complex decision-support queries [14] containing group-by, aggregation, outer joins, semi-joins, and nested sub-queries. Logical transformation rules in the optimizer transform a given query expression into a semantically equivalent query expression. The optimizers are equipped with these transformation rules to enable them to find an efficient execution plan for a given query by applying the transformation rules selectively. However, even though verifying equivalence of two query expressions was never the goal of the query optimizer, the optimizer can be adapted to serve as a specialized query equivalence verifier by leveraging its logical transformation rules. Our query equivalence mechanism, which we call QO-Verify (described in Section 2), takes as input *two* SQL queries Q and Q'. It returns TRUE if it can find *at least one logical expression in common* between the set of logical expressions that are explored for Q and Q' using the transformation rules available to the optimizer, or UNKNOWN otherwise. This check is sufficient to prove equivalence since each transformation rule is known to be sound, i.e., applying a rule preserves semantic equivalence. We implement this technique for Microsoft SQL Server's query optimizer, which is developed using the Volcano/Cascades framework [20, 21]. Our technique leverages the *memo* structure, which compactly represents the space of logical expressions explored, to develop an efficient algorithm. In our experiments (see Section 4), we observe that using QO-Verify we could successfully verify equivalence in 37% of the rewrites suggested by LLMs. The median running time of QO-Verify across all its invocations is around 1.2 sec, which makes it usable for verifying equivalence of two SQL queries.

Not surprisingly, QO-Verify can be expected to verify equivalence only when the transformation rules required to transform Q to Q' already exist in the optimizer. In our experiments, we find examples of non-trivial LLM-suggested rewrites whose execution time is significantly faster than the original query, and whose results on the current database are identical to the original query; however the above equivalence check mechanism returns UNKNOWN for that pair. In such cases, it is worthwhile to ask the question if we could distill candidate transformation rules that can be proved to be correct (just like transformation rules that are part of today's optimizers) that together with existing transformation rules in the optimizer imply the equivalence of Q and Q'. Furthermore, if such distilled transformation rules are broadly applicable and of high value to cost-based query optimization, they may be candidates to be added to query optimizers in the future. In Section 3, we illustrate this possibility by presenting a few transformation rules that are not currently in Microsoft SQL Server but they contribute to significant improvements on real-world and benchmark queries. A common theme among the rules is that they attempt to reduce repeated computation of exact or similar expressions in the query. Like most rules, these rules are not always beneficial to apply, and must therefore be applied in a cost-based manner. In extensible query optimizers incorporating new rules is relatively straightforward, since a rule can be added without having to change the search algorithm [21], although mechanisms such as *guidance* and *promise* [14] would need to be set appropriately for applying these rules.

2 VERIFYING EQUIVALENCE USING THE QUERY OPTIMIZER

In this section we describe a technique, which we refer to as QO-Verify, for identifying if two SQL queries are semantically equivalent by extending the built-in capabilities of the query optimizer. Specifically, we rely on the fact that modern query optimizers use logical transformation rules to define the search space of alternative plans to consider for a query. Each rule is known to *preserve semantic equivalence* when applied. We note that there are other techniques that attempt to either prove that two queries are equivalent, e.g., QED, Veri-EQL, Cosette, SQLSolver [10, 15, 22, 39], or testing-based approaches that show that two queries are *not* semantically equal by generating counter-example databases, e.g., XData [8]. These techniques are complementary and can be used in conjunction with the equivalence checker described below.

Given two queries Q and Q', our goal is to verify if the two queries are semantically equivalent. Conceptually, this could be achieved by enumerating for each query, all logical expressions that could be generated by applying the transformation rules available in the optimizer. Then, if the two queries share *at least one* logical expression in common, we know that they are semantically equivalent since each transformation rule is known to be sound. If no expressions are found in common, then we do not know if the two queries are semantically equivalent or not. Therefore this approach provides a one-sided check for semantic equivalence.

While the above approach is sound, a straightforward implementation of this approach can be quite expensive since the space of all logical expressions that can be derived by applying transformation rules can be large. Therefore, the cost of enumerating logical expressions for the purposes of comparing with a logical expressions for the other query can be expensive. In this section we present an efficient technique for checking equivalence. We describe our technique in the context of a query optimizer built using Volcano/Cascades [20, 21], an extensible framework for query optimizers. Our implementation uses the Microsoft SQL Server query optimizer[6, 16, 17], which uses the Cascades framework. The QO-Verify algorithm, described in Section 2.2, leverages the memo, a key data structure used in Volcano/Cascades, for efficiency. The input and output of QO-Verify and its usage of the query optimizer is shown in Figure 1. We begin with a brief overview of the memo

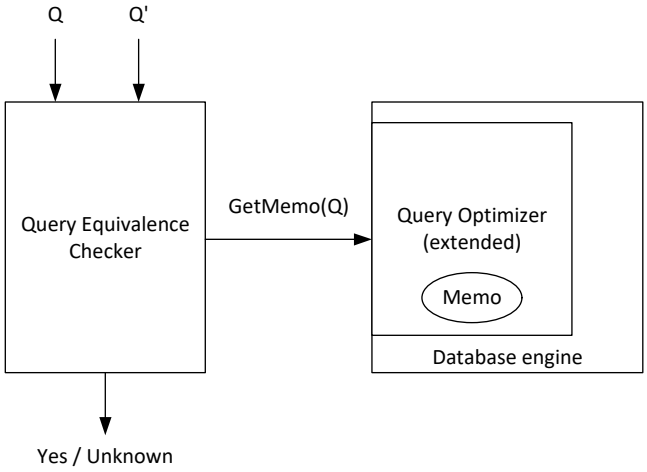


Figure 1: QO-Verify: Checking query equivalence using the query optimizer

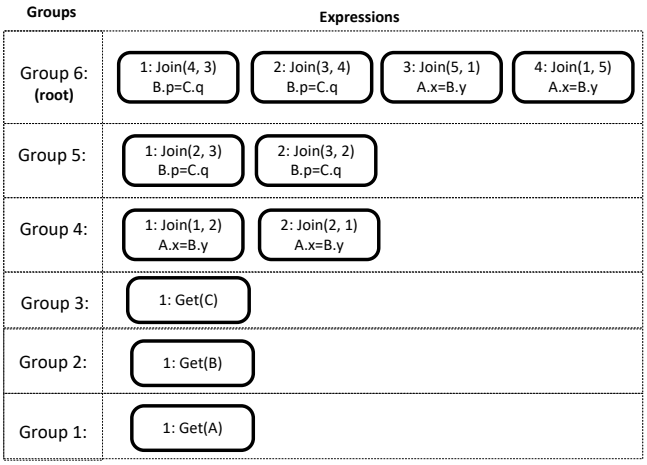


Figure 2: A simplified memo data structure showing groups and logical expressions for example Query 1.

in Section 2.1, and then describe the algorithm and implementation details in Section 2.2.

2.1 Memo overview

We provide a brief overview of the memo, a key data structure used in Volcano/Cascades to compactly represent the logical and physical expressions considered by the optimizer during its search. To illustrate the basic concepts of a memo we use the following example query. The memo for Query 1 containing the groups and the logical expressions are shown in Figure 2.

Query 1

```
SELECT * FROM A, B, C
WHERE A.x = B.y AND B.p = C.q
```

A *group* contains all logical and physical *expressions* that are semantically equivalent. A logical expression has a logical operator (e.g., Join) as its root and groups as its inputs (i.e., children). Thus, groups and expressions are mutually recursive, which enables compact representation of all expressions that the optimizer considers. Each group contains expressions explored by the optimizer for that group by applying transformation rules. Since we are interested with semantic equivalence (and not physical plans), in the figure we only show the groups and logical expressions.

A special group is the *root* group, which contains all logical expression corresponding to the input query. In our example memo, Group 6 is the root group. For convenience we number each expression in a group. For example, expression 6.1 is a *Join* with Group 4 as the left child and Group 3 as the right child, and predicate $B.p = C.q$. Group 4 represents all expressions for $A \bowtie_{A.x=B.y} B$, and Group 3 represents all expressions for C – in this case only one since it is a leaf node representing Get(C). Thus 6.1 in the memo represents two different complete logical expression for Query 1: Join(Join(A,B),C) and Join(Join(B,A),C).

Alternative logical expressions within a group are obtained by applying transformation rules based on known equivalences in SQL. For example, since an inner join is commutative and associative, the optimizer may apply logical transformation rules for join commutativity and join associativity to generate alternative logical expressions for an expression with inner join as the root. Modern query optimizers such as Microsoft SQL Server contain a rich set of transformation rules that were developed over the past few decades. These rules are crucial for optimizing complex, decision support queries containing operators such as inner and outer joins, semi-joins, group-by, aggregation, as well as correlated subqueries etc. We refer the reader to the monograph [14] for more details on the memo and how it is used during search.

2.2 QO-Verify: Optimizer-based equivalence checking

The pseudocode for our memo-based equivalence checker is shown in Algorithm 1. The algorithm takes two queries Q_1 and Q_2 as input and returns one of two values: True if the two queries are semantically equivalent or Unknown otherwise. The algorithm invokes the query optimizer once per query to obtain the memos M_1 and M_2 respectively. During this invocation of the query optimizer, we turn off cost-based pruning and timeouts (which in Microsoft SQL Server can be achieved by turning off task limits) to ensure that the optimizer explores the full space of logical expressions. We also turn off implementation rules to avoid wasted work. The algorithm then invokes MatchGroups, passing in the *root* groups of both memos. The function MatchGroups takes two groups g_1, g_2 one from each query, and returns true if *any* pair of expressions, one from each group are identical (i.e., they match exactly), or false otherwise. The correctness of this step follows from the fact that all expressions within a group are semantically equivalent to one another since each expression is derived by applying equivalence preserving transformation rules. The check for equivalence of two logical expressions is carried out by the function MatchExprs which takes as inputs two expressions e_1 and e_2. It checks that the operators corresponding to the root expressions of e_1 and e_2 are

Algorithm 1 QO-Verify: Optimizer-based Query Equivalence

```
Input: Two queries Q1, Q2.
Output: true if equivalent, else unknown.
1:  function Equivalent(Q1, Q2)
2:      M1 ← GetMemo(Q1)
3:      M2 ← GetMemo(Q2)
4:      return MatchGroups(M1.root, M2.root) ? true : unknown
5:  function MatchGroups(g1, g2)      ▷ Equivalent if any pair of
    expressions, one from each group, are equal
6:      for all e1 ∈ g1.exprs do
7:          for all e2 ∈ g2.exprs do
8:              if MatchExprs(e1, e2) then
9:                  return true
10:     return false
11: function MatchExprs(e1, e2)
12:     if e1.root ≠ e2.root or |e1.children| ≠ |e2.children| then
13:         return false
14:     for i ← 1 to |e1.children| do
15:         if not MatchGroups(e1.children[i], e2.children[i])
    then
16:             return false
17:     return true
```

identical and that each operator has the same arity i.e., number of children. If either of these conditions are not satisfied, the two expressions cannot be identical, and hence the function returns FALSE. Recollect from Section 2.1 that the children of an expression are memo groups. If both conditions are satisfied above, MatchExprs checks that *each* pair of corresponding child groups are identical to one another by invoking MatchGroups on the corresponding child groups. Note that the order of children is important, e.g., OuterJoin(A, B) is not equivalent to OuterJoin(B, A) in general, and therefore the order of children is preserved while performing the check.

Efficiency: The algorithm derives its efficiency from the following properties. First, if two logical expressions are identical, then in a top-down traversal of the two expressions, every corresponding node must be identical. Therefore, as soon as two corresponding nodes of the expressions disagree (Line 10), we can terminate the comparison of that pair of expressions. Second, since the algorithm performs a depth-first expansion of a logical expression, it can return as soon as one logical expression is found in common between the two memos. Third, unlike the straightforward approach that generates complete logical expressions, in this algorithm, we implicitly enumerate logical expressions of each memo while retaining the compactness of representation of the memo. Fourth, although not shown in the pseudocode for simplicity, by maintaining a cache of outcomes of matching for (e_i, e_j) pairs of expressions (besides expressions in the root group) of the query, it is possible to avoid the work associated with a repeated invocation of MatchExprs for the same pair of expressions. Similarly, once MatchGroups for a pair of groups (other than the root group) has been completed, its result can be cached and reused for any future invocations of the same pair of groups. Such caching is effective since a given expression typically is a child of many other expressions. For example, the expression Get(A) in Group 1, is a child of four other expressions (4.1, 4.2, 6.3 and 6.4) in the memo in Figure 2. The caching described above trades off additional memory for reduced work. In Section 4 we report the performance and memory usage of this mechanism across the queries we experimented with.

Accuracy and Coverage: In order to return TRUE the above algorithm requires the *exact match* of a pair of logical expressions, one from the root group of M_1 and the other from the root group of M_2. In all other cases, it returns UNKNOWN. It does not produce any false positives. The *coverage* of the above algorithm, i.e., the fraction of truly semantically equivalent cases for which it returns True, depends on a few factors. First, the set of transformation rules needed to transform an expression of Q_1 to an expression of Q_2 (or vice versa) must exist as part of the transformation rules implemented in the query optimizer. As discussed in Section 3, we have found examples of LLM-based rewrites where the rules required for establishing equivalence are not available in the optimizer. Second, for efficiency, many optimizers do not always apply every transformation rule that is applicable to an expression. Volcano/Cascades uses the notion of *guidance* to decide which rules are applicable for a given logical operator and *promise* to to heuristically decide which subset of applicable rules to apply for an expression. Hence by turning off these mechanisms, we can ensure that all relevant rules are applied, thereby potentially increasing coverage. Third, when the memo is generated, the implementation rules and traditional cost-based pruning used by query optimizers should be turned off. This allows the optimizer to explore logical expressions that may otherwise get pruned and thereby adversely affect coverage.

Implementation Notes: We implemented our technique in Microsoft SQL Server. We extended the query optimizer to export the memo after query optimization, e.g., similar to [7], and implemented the equivalence checking part of the algorithm in the client. Second, the checks in Line 10 in MatchExprs are operator specific. Consider the case where e_1 is the expression 4.1 in Figure 2, which is an inner join operator with predicate A.x=B.y. When comparing equality with e_2 from the other memo, the comparison logic would need to check if both logical operators are identical (inner join), *and* the join predicates are identical. Third, prior to memo creation, the optimizer performs several normalization steps, such as constant folding (e.g., A.x < 2+5 $\implies$ A.x < 7) and applying substitution rules. It also applies certain rules that it considers as always beneficial, e.g., decorrelation[16]. Such normalization reduces the variations possible between the logical expressions of the two queries, and therefore aids in improving the coverage of equivalence checking. In general, the scope of normalization can be broadened to aid in equivalence checking. For example, consider the same scalar expression (e.g., used in an aggregate function) which is written differently in the two queries: $(a + b + c)$ vs. $(b + a + c)$. These expressions are represented as trees. We can therefore increase coverage by normalizing the expression trees prior to the optimization.

Relationship to plan forcing APIs: Microsoft SQL Server exposes a functionality called USE PLAN, which is used to force a plan for a given query. Similar functionality is available in other database engines, e.g., SQLPlanBaselines in Oracle[34] and abstract plans

in Sybase[3]. In a USE PLAN call, the optimizer is given a query and a plan. It starts with the query and its goal is to find the given plan by applying sound algebraic transformation and implementation rules. Therefore, if the given plan is found by the optimizer's search, we know the plan is valid for that query. We note that it is possible to modify USE PLAN to take in only a logical expression corresponding to Q_2 (rather than a physical plan) and attempt to force that logical expression for Q_1. Such a modified version of USE PLAN could be used for equivalence checking. However, that approach is more restrictive compared to QO-Verify in the sense that it attempts to find if a *specific* expression of Q_2, namely application of the initial logical expression, can be reached via application of transformation rules starting from Q_1. Therefore, this approach can be viewed as a special case of our equivalence mechanism with a single logical expression in memo M_2. However, in cases where generating the memo for one of the queries is too expensive, such a modified version of USE PLAN can be used instead.

3 DISTILLING CANDIDATE TRANSFORMATION RULES FROM QUERY REWRITE EXAMPLES

Even when a query Q and its rewrite Q' are semantically equivalent, we may not always be able to verify the semantic equivalence of Q and Q' using the optimizer-based semantic equivalence mechanism QO-Verify (Section 2). This could be for a number of reasons. For example, although the optimizer transformation rules in many cases can help reason with unique aspects of SQL expressions that SMT solvers struggle with (e.g., nesting and aggregations), optimizers do not have all the general purpose theorem proving capabilities of SMT solvers. Another class of scenarios where a given optimizer may fall short in proving equivalence of Q and Q' is if the optimizer does not have all logical transformation rules that would be necessary to prove the equivalence of Q and Q'. Consider the case when QO-Verify fails to prove equivalence, but a DBA or a developer manually verifies that Q and Q' are indeed equivalent. Then, the pair (Q, Q') presents the *query optimizer developers* of that database engine an opportunity to distill *candidate* transformation rule(s) that might be missing in their optimizer that *if added* to the optimizer would result in QO-Verify inferring equivalence of Q and Q'[1]. Of course, before such candidate transformation rules are added to the optimizer, their broad applicability beyond proving equivalence of Q and Q' must be ascertained. Although the above approach of distilling such candidate transformation rules is manual, it is nonetheless beneficial in expanding the capabilities of the optimizer.

In the LLM-based rewrites we obtained through our experiments that led to significant reduction in elapsed time compared to the original query, we considered only those rewrites which generated the same result as the original query on the given database. We refer to these as *candidate* rewrites. We manually examined several candidate rewrites, and we observe two classes in them. The first class involves application of well-known transformations used by cost-based query optimizers to improve query performance. Two examples are: (1) A rewrite in which a group-by operator is pushed below UNION ALL. This rule has been implemented in the IBM DB2 database engine [23]. (2) A rewrite that replaces a UNION ALL of multiple group-by queries on the same expression with a GROUPING SETS clause over the same expression [29]. Even those these rules are well known, these rules are not among the logical transformation rules in Microsoft SQL Server. In the second class, the significant reduction in elapsed time of the rewrite is driven by the use of new transformation rules that represent variations of well-known transformations. In the rest of this section, we present three *candidate* transformation rules we derived manually based on candidate rewrites on Microsoft SQL Server. In each case, the QO-Verify mechanism returned Unknown for the query pair. These rules share a common characteristic that they apply to expressions involving *comparisons of multiple aggregate expressions* defined over a common sub-expression or similar sub-expressions, a pattern that is common in decision support applications [24]. As discussed in Section 5, these rules share similarity with well-known transformations [5, 25, 35, 46].

The methodology we followed to derive these candidate transformation rules was to examine the query and its rewrite, and manually distill from it a pair of simplified query expressions which embodies the transformation rule. We then created an algebraic representation of the transformation rule from the pair of query expressions. In this paper, we have not generalized the rules so derived beyond what we observed in the rewrite, e.g., by broadening the class of predicates supportable in the rule in Section 3.2 or extending the set of aggregates in Section 3.3. In practice, such generalizations may need to considered prior to inclusion of the rules into the query optimizer. Similar to most transformation rules in the optimizer, the rules we describe below must be applied by the optimizer in a cost-based manner.

3.1 Join of Group-By Subqueries ⇔ Window Functions

In one real-world query on the REAL-1 database, we observed a group-by subquery on a set of columns C, as well as multiple subqueries with group-by on *subsets* of C. The top-level query then joins the results of these subqueries, in order to compare aggregates produced by the subqueries.

Query 2

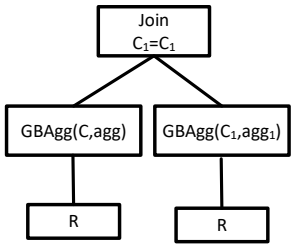
```
SELECT D2.p2, D2.s2, D1.q1, D2.q2
FROM
 (SELECT l_partkey as p1,
    SUM(l_quantity) q1
    FROM lineitem
    GROUP BY l_partkey) D1,
  (SELECT l_partkey as p2, l_suppkey as s2,
    SUM(l_quantity) q2
    FROM lineitem
    GROUP BY l_partkey, l_suppkey) D2
WHERE D1.p1 = D2.p2
```

Query 3

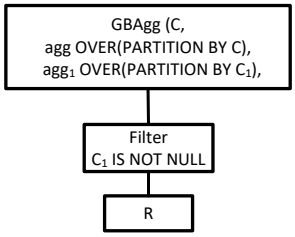
```
SELECT DISTINCT
    l_partkey as p2,
    l_suppkey as s2,
    SUM(l_quantity) OVER(PARTITION BY l_partkey) as q1,
    SUM(l_quantity) OVER(PARTITION BY l_partkey, l_suppkey) as q2
FROM lineitem
```

We show in Query 2 a simplified example of the real-world query expressed on the well-known TPC-H schema. The simplified

[1]A candidate transformation rule so discovered can also be a macro rule e.g., for optimizing star queries [17], as the addition of such rules facilitates search for equivalence.



(a) Joins over multiple group-by subsets.



(b) GBAgg operator with window functions.

Figure 3: R is an arbitrary relational expression. C is a set of columns of R. $C_1 \subseteq C$.

.

version of the rewritten query generated by the LLM is shown in Query 3. The rewrite uses window functions partitioned on C and C_1 respectively. In the real-world query, the corresponding rewrite executed 1.9× faster.

Figure 3 shows the candidate algebraic transformation rule we inferred from the above rewrite. R is any relational expression and $C_1 \subseteq C$, and C is not empty. Note that $GBAgg(X, a_1, a_2 \dots)$ is the group-by-aggregation operator, where X is the set of group-by columns from the input relation and $a_1, a_2 \dots$ are aggregate expressions (e.g., SUM(l_quantity)). agg and agg_1 are deterministic aggregate functions, e.g., SUM, MIN, MAX. Intuitively, when the work done in computing relation R is significant (e.g., R itself is a complex expression with multiple joins), the transformation to a window function on R may be more efficient. For example, in the real-world query, the optimizer spools the result of R in the rewritten query, thereby contributing to reduced execution cost, whereas it does not in the original query.

To understand the candidate rule, consider the LHS expression in Figure 3. Since C is a superset of C_1, the join in the expression is a many-to-one join. Observe that the join eliminates any rows where C_1 is NULL. Hence the final result of the expression is the result of $GBAgg(C, agg)$, with cardinality equal to the number of distinct values in C where C_1 is NOT NULL. The Join extends the result with an additional column agg_1, whose values for a given row equals the aggregate corresponding to the joining value C_1. Examining the RHS expression, the Filter eliminates all rows from R where C_1 is NULL, i.e., any row that passes the Filter must have each column in C_1 NOT NULL. By the semantics of the GBAgg operator with window function agg OVER(PARTITION BY C), the RHS query has the same set of columns C and the column agg in the output. Since $C_1 \subseteq C$, for any value $v \in C_1$, the window function agg_1 OVER(PARTITION BY C_1) produces the same aggregate value for all partitions of C whose C_1 value is v. The GBAgg in the RHS expression eliminates all duplicates in the result, thereby also ensuring the same output cardinality of the LHS expression.

3.2 Self-join of Group-By ⇔ CASE

Consider Query 4, a simplified version of Q74 in the TPC-DS benchmark. This query finds the total sales for each customer who had sales in *both* the years 2000 and 2001. Such a query that compares aggregate metrics across years (or stores or products) is commonly used in data analytics [24].

Query 4

```
with year_total as (
 select c_customer_id customer_id
       ,d_year as year
       ,SUM(ss_net_paid) total_sales
 from customer
     ,store_sales
     ,date_dim
 where c_customer_sk = ss_customer_sk
   and ss_sold_date_sk = d_date_sk
   and d_year in (2000,2000+1)
 group by c_customer_id
         ,d_year
)
  select t_s_secyear.customer_id,
  t_s_firstyear.total_sales as s2000,
  t_s_secyear.total_sales as s2001
 from year_total t_s_firstyear
     ,year_total t_s_secyear
 where t_s_secyear.customer_id = t_s_firstyear.customer_id
         and t_s_firstyear.year = 2000
         and t_s_secyear.year = 2000+1
```

Consider Query 5 below which is semantically equivalent to Query 4, but executes 3.6× faster on Microsoft SQL Server. We note that the self-join over the grouped relation *year_total* in Query 4 is replaced by a *GBAgg* operator over the grouped relational *year_total*, but computing two *conditional aggregates* using CASE statements, conditioned on the year. Observe that the conditions in the HAVING clause are necessary to ensure that only customers with sales in *both* years appear in the final results of Query 5. The transformation avoids computing R multiple times and hence may be faster when computing R is expensive. The candidate transformation rule extracted by analyzing the above pair of queries is shown in Figure 4. R is any relational expression and agg is any deterministic aggregate. a and b are sets of columns in R.

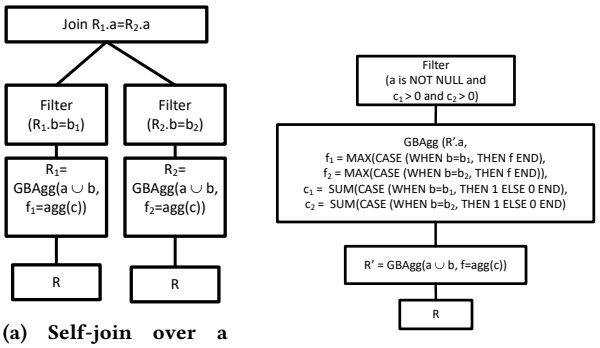
Query 5

```
with year_total as (
 select c_customer_id customer_id, d_year as year
    ,SUM(ss_net_paid) total_sales
 from customer
    ,store_sales
    ,date_dim
 where c_customer_sk = ss_customer_sk
    and ss_sold_date_sk = d_date_sk
    and d_year in (2000,2000+1)
 group by c_customer_id, d_year
    ),
v1 as (
    select
        customer_id,
        max(case when year = 2000 then total_sales end) as s2000,
        max(case when year = 2001 then total_sales end) as s2001
    from year_total
    group by
        customer_id
        having sum(case when year = 2000 then 1 else 0 end) > 0 and
         sum(case when year = 2001 then 1 else 0 end) > 0
    )
select
    customer_id, s2000, s2001
from v1
```

2

Consider the LHS of the rule. R_1 and R_2 are identical expressions. Since the join requires R.a values to match and the filters require

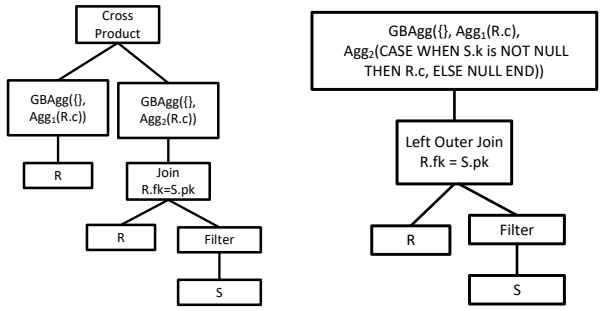
[2]We note that MIN can be used instead of MAX without altering the semantics since .



(a) Self-join over a grouped relation with filters.

(b) Group-by with conditional aggregate over grouped relation.

Figure 4: Transformation of self-join over grouped relation R_1 (=R_2) to group-by with conditional aggregate over grouped relation.



(a) The Join below Agg_2 is over a subset of R.

(b) Left Outer Join and conditional Agg_2 using CASE

Figure 5: Comparison of two scalar aggregates. The join between R and S is a Foreign-Key (R) Primary-Key (S) join.

$R_1.b = b_1$ *and* $R_2.b = b_2$, in the LHS query: (1) The result only contains those a values that have one or more rows in R with $b = b_1$ *and* one or more rows in R with $b = b_2$. (2) The result contains *exactly one row* per $R.a$ value, thereby making $R.a$ a key of the output. The values of $R_1.f$ and $R_2.f$ output correspond to the aggregate $\text{agg}_1(R.c)$. Now, we consider the RHS of the equation. The topmost *GBAgg* expression has $R.a$ as the key. The aggregate expression f_1 only outputs the value of $R.f$ that corresponds to $R.b = b_1$. Since (a, b) is a key of R, there can be at most one such value in R. For all other values of $R.b$ the CASE statement returns NULL, and the semantics of the MAX aggregate over a NULL value is to return NULL. Since there is at most one non-NULL value per group, f_1 returns the $agg(R.c)$ for the rows satisfying $R.b = b_1$ for that particular value of $R.a$. A similar argument applies for f_2. Note that this *GBAgg* expression may return rows where either f_1 (resp. f_2) is NULL when for a particular value of $R.a$, there are no rows in R where $R.b = b_1$ (resp. $R.b = b_2$), whereas the LHS does not. Therefore, c_1 and c_2 are required to count the number of rows for each value of a where $b = b_1$ and $b = b_2$ respectively. The Filter operator then discards all rows where either $c_1 = 0$ or $c_2 = 0$. Finally, the Filter (a IS NOT NULL) is required for the general case where $R.a$ is NULL since the Join in the LHS eliminates all rows where $R.a = NULL$.

3.3 Subsumed Scalar Aggregate Subquery ⇔ Scalar Aggregate over Outer Join

A common pattern in analytical queries is to compare an aggregate metric, e.g., *total sales*, with the same metric under a more restrictive condition, e.g., total sales *during a promotion*. An example of this is TPC-DS Q61, a simplified version of which is shown in Query 6. Observe that the rows over which the aggregate of the first subquery is computed is subsumed by the rows over which the aggregate of the second query is computed. This is because the subqueries are identical except for an additional K-FK join with the *promotion* relation and filter predicates on the *promotion* relation. Such a join and filter cannot increase the number of rows compared to the FK relation in the second subquery. Query 7 is a simplified version of the rewrite produced by the LLM. The rewritten query for the original TPC-DS Q61 executes 1.8× faster on Microsoft SQL Server.

Query 6

```
select promotions,total
from
  (select sum(ss_ext_sales_price) promotions
   from  store_sales
        ,store
        ,promotion
        ,date_dim
        ,item
   where ss_sold_date_sk = d_date_sk
   and   ss_store_sk = s_store_sk
   and   ss_promo_sk = p_promo_sk
   and   ss_item_sk = i_item_sk
   and   i_category = 'Jewelry'
   and   (p_channel_dmail = 'Y' or p_channel_email = 'Y'
   or p_channel_tv = 'Y')
   and   s_gmt_offset = -6
   and   d_year = 1999 and d_moy  = 12) promotional_sales,
  (select sum(ss_ext_sales_price) total
   from  store_sales
        ,store
        ,date_dim
        ,item
   where ss_sold_date_sk = d_date_sk
   and   ss_store_sk = s_store_sk
   and   ss_item_sk = i_item_sk
   and   i_category = 'Jewelry'
   and   s_gmt_offset = -6
   and   d_year = 1999 and   d_moy  = 12) all_sales
```

Query 7

```
select
  sum(case when p.p_promo_sk is not null
  then ss.ss_ext_sales_price else 0 end) as promotions,
  sum(ss.ss_ext_sales_price) as total
from store_sales ss
join date_dim d
  on ss.ss_sold_date_sk = d.d_date_sk
join store s
  on ss.ss_store_sk = s.s_store_sk
join item i
  on ss.ss_item_sk = i.i_item_sk
left join promotion p
  on ss.ss_promo_sk = p.p_promo_sk
    and (p.p_channel_dmail = 'Y' or p.p_channel_email = 'Y'
    or p.p_channel_tv = 'Y')
where  i.i_category = 'Jewelry'
  and s.s_gmt_offset = -6
  and d.d_year = 1999 and d.d_moy = 12
```

Figure 5 shows an algebraic representation of the candidate transformation rule. Intuitively, the RHS can be more efficient when the expression R is expensive to compute. The LHS expression produces a single row as output containing two scalar aggregates. Agg_1 is computed on the result of R, and Agg_2 is computed on a subset of rows of R that join with $Filter(S.a)$ via a Primary-Key Foreign-Key join. Due to the semantics of Left Outer Join (LOJ) in the RHS expression the inputs to the GBAgg operator has the same cardinality as R and all rows from R are preserved in the result of the LOJ. The column $S.k$ from S, which is part of the output of the LOJ has a non-NULL value exactly for the rows that pass the Filter(S.a) and join with $R.fk$.

4 EXPERIMENTS

The goal of our experiments are to: (1) Evaluate the effectiveness of rewrites generated by LLMs on real-world queries over enterprise databases, and compare with the results on benchmark queries. (2) Evaluate the coverage and overheads of the semantic equivalence checker using the query optimizer presented in Section 2.2 for query rewrites generated by LLMs.

4.1 Setup

Databases and queries: We perform our evaluation on several real-world queries on four Microsoft SQL Server customer databases, industry benchmarks TPC-DS[31], TPC-H with skewed data distribution, as well as the synthetic optimizer benchmark JOB [26] as shown in Table 1. REAL-1, REAL-2, REAL-3 and REAL-4 are data warehouses for analytics for four different Microsoft customers: a cosmetics retailer, a software company's sales, a book retailer's sales, and an ERP application respectively. The real-world queries exercise logical SQL constructs such as inner and outer joins, group-by and aggregation, union all, subqueries, views, and Common Table Expressions (CTEs). For the real-world queries, the number of tables in a query varied between 1 and 28 as shown in the table. To ensure that the baseline for our experiments consists of a well-tuned physical design, as one would expect in production queries, for each database we tune indexes using Microsoft SQL Server's Database Tuning Advisor (DTA) [1]. We provide all the queries for that database as the input workload, and implement DTA's recommendations.

Database	Number of queries	Min, Max, Median number of tables in query
REAL-1	39	1, 8, 3
REAL-2	40	5, 28, 16
REAL-3	25	1, 8, 4
REAL-4	12	4, 14, 10
TPC-DS	99	1, 13, 5
JOB	113	4, 15, 8
TPC-H 10GB z=1	22	1, 7, 3

Table 1: Databases and queries

4.2 Evaluation of LLM-Based Rewriting

Prompts/LLM used: We use three prompts borrowed from prior work[12, 28]: (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. 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 [30]. 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 use the GPT-4o-mini and GPT-5 models [32, 33] from OpenAI in our experiments.

Execution and retry: We invoke the LLM with each different prompt and execute the rewrite returned. We measure the elapsed time of cold runs. If the rewrite is not valid SQL, or if the elapsed time of the resulting rewritten query is not significantly faster than the original query (we use a threshold of 2×), we retry the same prompt a maximum of 2 additional times for a maximum of 3 total LLM invocations. During a retry, we append to the prompt any errors obtained for the prior invocations. We also check if the results (ignoring ordering of rows) of executing the original and the rewritten query on the given database are exactly the same. If so, we refer to such rewrites as *candidate* rewrites. We discard any rewrites where the results differ. The fraction of rewrites which differed in the results were around **32%**, which points to another limitation of LLM-based rewriting. In this paper, we only report candidate rewrites. We obtained 3100+ candidate rewrites in total across all databases using the above methodology. The percentage of candidate rewrites broken down by database are: REAL-1 . . . REAL-4: 13%, 10%, 5% and 5% respectively, and TPC-DS, JOB and TPC-H: 24%, 38% and 5% respectively.

Improvements of candidate LLM-based rewrites: Figure 6 shows a scatter plot of elapsed time on the original query on the x-axis and the elapsed time of the rewrite on the y-axis. We show all 3100+ candidate rewrites. Since there is a wide range of elapsed times across queries, we use a log-log scale. We observe many points distant from the diagonal, indicating that many rewrites are significantly faster than the original query (points below the diagonal), and many other rewrites are significantly slower than the original query (points above the diagonal).

In Figure 7 we show the distribution of improvement in elapsed time of the *best* rewrite for each query, i.e., the rewrite with the lowest elapsed time. We show this distribution across all candidate rewrites (blue line), all rewrites for all benchmark queries (green line), and all rewrites for queries on real databases (orange line). We observe that the **median** query improves by **38%, 46% and 29%**

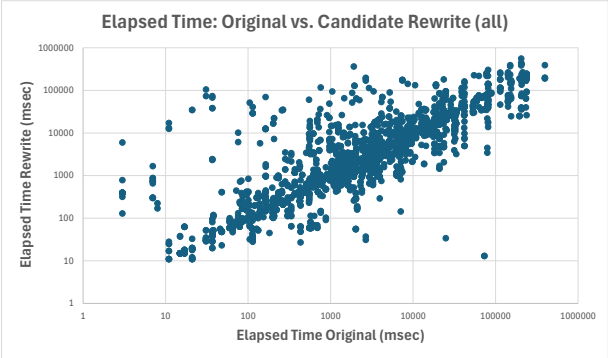


Figure 6: Scatter plot of elapsed time (cold run) for all candidate rewrites.

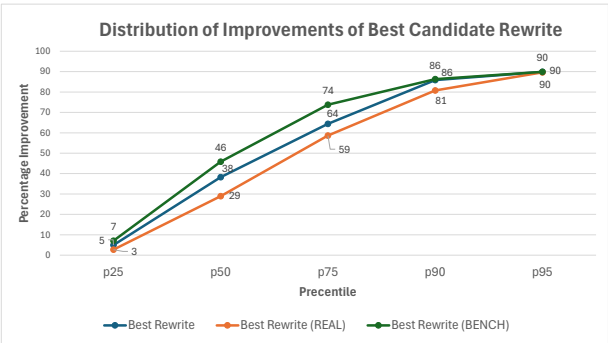
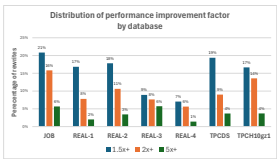
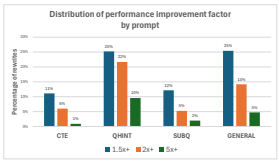


Figure 7: Improvement percentage at different percentiles for real and benchmark queries.



(a) Breakdown of improvements by database



(b) Breakdown of improvements by prompt

Figure 8: Percentage of rewrites with a given improvement factor in elapsed time.

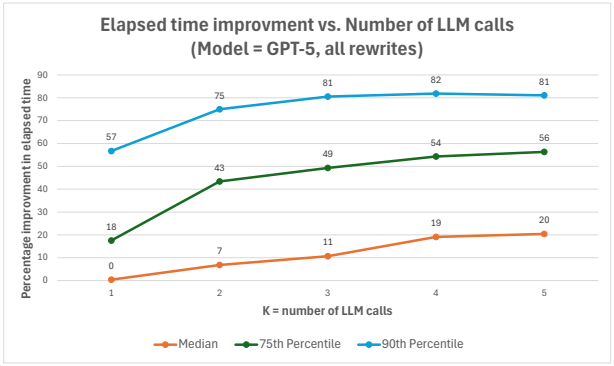


Figure 9: Percentage improvement vs. Number of rewrites needed

respectively. At higher percentiles the differences in improvements between real and benchmark queries are even less. For example, for 10% of the queries (i.e., at **P90**), the improvements are **86%, 86% and 81%** respectively. Overall, we observe that candidate LLM-rewrites provide meaningful improvements for both benchmark and real queries, which suggest some capabilities of generalization by LLMs.

Figure 8a shows the percentage of rewrites that improve the elapsed time by a factor 1.5× or more, 2× or more, and 5× or more for each database. We observe that all four real databases show rewrites with noticeable improvements. Figure 8b shows a similar breakdown, but by the type of prompt used. We observe that the largest fraction of rewrites improve when using the QHint and General prompts, whereas for the other two prompts, the corresponding fractions are noticeable but relatively smaller.

Variance in quality across LLM invocations: One important issue of LLMs for query rewriting in practice is the *variation in quality* of rewrite across multiple LLM invocations. This is important since LLM invocations are costly, both in terms of resources used and monetary cost. In this experiment, we aim to determine how the improvement in performance varies as we make more invocations to the LLM to produce rewrites. We consider all candidate rewrites generated by the GPT-5 model and we order the rewrites in *random* order. We then choose K rewrites in the above order, where $1 \leq K \leq 5$. For each value of K, we plot the improvement percentage of the best rewrite among the first K (see Figure 9). For the median query, the improvements are modest even with $K = 4, 5$, whereas they are significant at higher percentiles (75-th and 90-th percentiles). This experiment shows that given this intrinsic variability in quality of LLM rewrites, to obtain significant improvements, e.g., 2× or more improvement at P75, three or more rewrites may be necessary in practice. Finally, two other factors can further increase cost of using LLMs for query rewriting: (1) Tools that use LLM-based rewriting to tune a query would need to also execute the rewrite to be confident of the benefit, and hence would add overheads beyond LLM invocation. (2) As noted earlier, a significant fraction of rewrites were not even *candidate* rewrites, hence the actual number of invocations needed can be even higher in practice. Hence, it is worth studying if the prompts can be improved, or other models with higher accuracy be used to reduce the number of LLM-based rewrites needed in practice.

Comparing quality of GPT-5 vs. GPT-4o-mini: For each query, we plot the elapsed time of the best verified rewrite obtained using GPT-5 (x-axis) vs. the elapsed time of the best verified rewrite obtained using GPT-4o-mini. We see that for many queries, the two models produce the same rewrite or rewrite with similar performance – these are the points along or close to the diagonal. However, there are many queries where one of the models produces at least one significantly better rewrite than the best rewrite of the other. Overall, however, we do not observe a statistically significant difference in the quality between these two models. Understanding the relative

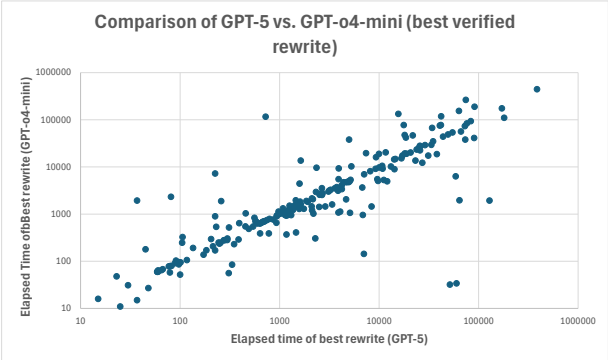


Figure 10: Scatter plot of elapsed time of rewrites using GPT-5 and GPT-4o-mini

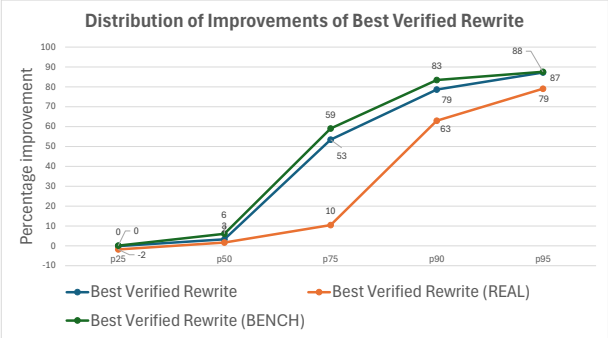


Figure 11: Improvement percentage at different percentiles.

effectiveness of different LLM models for query rewriting, including other models not studied here, is an important area of future work.

4.3 Semantic Equivalence Checking Using the Query Optimizer

We use the semantic equivalence checker QO-Verify described in Section 2.2 on each candidate rewrites. This checker can either return TRUE if it can show that the two queries are equivalent, or UNKNOWN otherwise. It cannot disprove equivalence. Out of the 3100+ candidate LLM-based rewrites, we are able to verify equivalence for 1150+ rewrites using QO-Verify. Hence the *coverage*, i.e., percentage of candidate rewrites we are able to verify is ≈**37%**.

Figure 11 shows the distribution of improvement in elapsed time of the *best verified* rewrite for each query, i.e., the rewrite with the lowest elapsed time that we are able to verify using QO-Verify. We show this distribution across all verified rewrites (blue line), all verified rewrites for all benchmark queries (green line), and all verified rewrites for queries on real databases (orange line). We observe that the improvements for the **median** query are not significant, all at or below**6%**. However, we see that at higher percentiles improvements are significant. For example, for 10% of the queries (i.e., at **P90**), percentage improvements are **79%, 83% and 63%** respectively. Overall, this technique shows the promise of QO-Verify on real and benchmark queries.

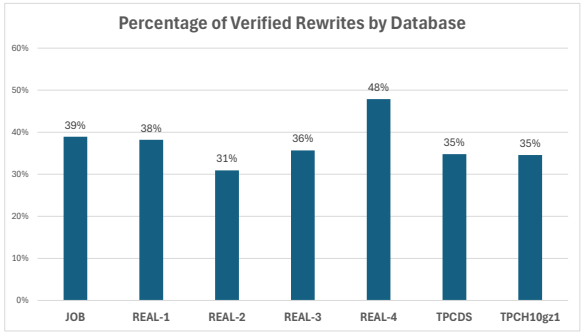


Figure 12: Percentage of rewrites verified by database.

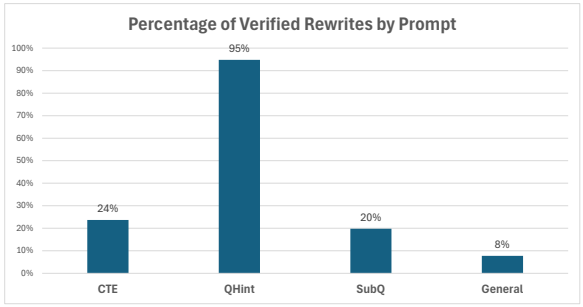


Figure 13: Percentage of rewrites verified by prompt.

In Figure 12 we show the fraction of rewrites verified by database; this varies between 31% and 48% depending on the database. The figure also shows that fraction of rewrites verified on real-world and benchmark queries are similar. Figure 13 shows that the coverage varies significantly by prompt. Query hint prompts have the highest coverage, which is not surprising since many hints, e.g., physical operator hints, do not alter the space of logical expressions explored by the optimizer, and our verification mechanism disables cost-based pruning. SUBQ and CTE prompts result in coverage of approximately 1 in 5, and 1 in 4 respectively. Intuitively, these prompts tend to make localized changes to the original query by refactoring common subexpressions. In contrast, the GENERAL prompt produces rewrites where the SQL generated tends to be most different when compared to the original query. These results suggest that by constraining the changes that the LLM is allowed to make, it may be possible to obtain higher coverage.

Overheads: We report the running time and memory overheads of using the QO-Verify mechanism (Section 2.2). The algorithm in Section 2.2 can be tuned to trade-off overheads with coverage. We tried two settings: (a) Specify no task limit for the query optimizer when generating the memo. (b) Use the optimizer's default task limit setting when generating the memo. For (a) the median and P90 times to obtain the memo of a single query are 0.57 sec and 17.5 secs respectively; and the median and P90 times for the matching algorithm are 0.03 sec and 5.4 sec respectively. The **median** time for QO-Verify overall is around **1.2 sec**. The median and P90 memory sizes for the memo (exported in XML format from the query optimizer) are 0.33 MB and 4.4 MB respectively. For (b) the

median and P90 times to obtain the memo of a single query are 0.41 sec and 3.5 secs respectively; and the median and P90 times for the matching algorithm are 0.02 sec and 0.48 sec respectively. The **median** time for QO-Verify overall is around **0.85 sec**. The median and P90 memory sizes for the memo are 0.25 MB and 2.2 MB respectively. We observed that the drop in coverage from (a) to (b) was around 2 percentage points. This experiment shows overall that it is practical to use the equivalence checking mechanism for SQL query tuning scenarios, and that by controlling the time spent by the optimizer in generating the memo, applications can potentially trade-off coverage for efficiency.

5 RELATED WORK

LLM-based query rewriting: GenRewrite[28] 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. Similarly, R-Bot [38] uses multi-step generation of rewrite to reduce the likelihood of incorrect LLM-generated rewrites, and LLM-R^2 [27] uses the LLM to decide which rules to apply for a given query when generating a rewrite. LITHE [12] 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. The above techniques demonstrate that LLMs can generate rewrites on complex queries. They evaluate on PostgreSQL and observe significant improvements in elapsed time on TPC-DS [31], TPC-H, DSB [13], and JOB [26], queries. LITHE also evaluates on a small number of queries on a private customer database and observed a geometric mean reduction in execution time of 3.3×. Our work is complementary in two key aspects. First, we broaden empirical evaluation on real-world queries. Second, we present a technique for automatically verifying semantic equivalence of the generated query rewrite. Our mechanism enables applications to close the tuning loop without human intervention. LLMSTEER[2] 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 rewriting using rules: The body of work on query rewriting using rules includes discovering rules for query rewriting e.g., [40] and generation of rules based on rewriting examples [4]. While these approaches can generate new rewrite rules, they are limited either by the manual effort required to define rewrite rules or the difficulty of automatic verification of candidate rules due to limitations of SQL equivalence checkers. As noted in [12], a state-of-the-art query rewriting technique [40] is unable to generate rewrites for TPC-DS queries. There is another line of work for applying rules to produce valid rewrites e.g., [44],[27], [38]. We note that some of these techniques use LLMs to improve efficiency of search by identify which subsets of rules to apply and in what order. All of these techniques are orthogonal to our main focus: to develop a mechanism for equivalence checking using the query optimizer.

Equivalence of SQL queries: Several techniques have been developed for proving equivalence of two SQL queries e.g., UDP[9], SPES[43], Cosette[10], QED[39], SQLSolver[15], VeriEQL[22] and WeTune[40]. One common approach is to convert each query into a first-order logic formula, and use SMT solvers (e.g., [11]) 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 [12], these methods fail on all TPC-DS rewrites generated by LLMs. We experimented with QED[39] and found that it is unable to verify any of the LLM-generated rewrites that we have obtained for the real-world queries. In Section 2.2 we noted the relationship of our query equivalence mechanism QO-Verify to the plan forcing (USE PLAN) functionality. LITHE [12] uses plan forcing, but not for the purposes of verifying semantic equivalence; rather they use plan forcing for parameterized queries to obtain the optimizer estimated costs of the original query's plan and rewritten query's plan for different instances of the parameterized query.

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[37], which uses LLMs for performance debugging, DBG-PT[18] for diagnosing performance regressions, $\lambda - Tune$ [19] for tuning database knobs, DB-GPT[45] for index tuning. An alternative approach to using LLMs off-the-shelf is proposed in [41] to train foundation models to learn representations of data, logical and physical query plans for use in multiple tuning tasks.

Related transformation rules: The three transformation rules presented in Section 3 share a common characteristic that they apply to queries that *compare results of multiple aggregate expressions* defined over a common sub-expression or similar sub-expressions. The above rules are closely related to rules identified in prior work, but they cover new cases. For example, compared to the idea of query fusion presented in [5], the Compare operator in COMPARE [36], reducing the cost of shuffle in a big-data system by sharing computation needed for multiple expressions in Blitz [25, 35], and sharing computation across windowed aggregates in a streaming engine [42], the rule in Section 3.1 can be viewed as extending the scope of such "query fusion" to cases where the group-by columns of sub-expressions are not identical.

6 CONCLUSION

Our broader empirical studies over enterprise workloads further reinforces the conclusion from earlier papers that LLM-suggested query rewriting may be valuable for performance tuning. However, the limiting factor for large-scale and automated adoption of query rewriting using LLMs is verifying that the rewritten query is semantically equivalent. In this paper, we introduced an equivalence checker that leverages query optimizers themselves for equivalence checking. Our approach makes it possible to do performance tuning through LLM-suggested query rewrites without human intervention or manual checking for a fraction of LLM-suggested rewrites. It is an open question as to whether the coverage of our equivalence checker can be augmented by leveraging SMT solver based methods for cases where our approach return Unknown. Another promising direction is to use techniques for testing, including formal methods, to generate counter-examples that prunes

out erroneous rewrites efficiently. Developing techniques for automated query rewriting that produces verified and performant rewrites in a cost-efficient manner by leveraging LLMs, equivalence checkers and testing techniques judiciously is important to fully realize the benefits of LLM-based rewriting. Finally, developing tools that aid distilling new candidate transformation rules from a database of queries and their manually verified rewrites will be interesting. Indeed, with suitable anonymization, the community can pull together to build such a database of pairs of queries and their manually verified rewrites.

7 ACKNOWLEDGMENTS

We thank our colleagues at Microsoft Kukjin Lee, Arnd Christian Konig, and Nico Bruno for their significant contributions in ensuring correctness and improving efficiency of the query equivalence mechanism in the optimizer. We are also greatly appreciative of our colleagues Vassilis Papadimos, Kaushik Rajan, Xiaoying Wang, and Wentao Wu who gave us valuable feedback.

REFERENCES

[1] Sanjay Agrawal, Surajit Chaudhuri, Lubor Kollar, Arun Marathe, Vivek Narasayya, and Manoj Syamala. 2005. Database tuning advisor for microsoft sql server 2005. In *Proceedings of the 2005 ACM SIGMOD*. 930–932.

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

[3] Mihnea Andrei and Patrick Valduriez. 2001. User-Optimizer Communication Using Abstract Plans in Sybase ASE. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 29–38.

[4] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2023. Querybooster: Improving SQL performance using middleware services for human-centered query rewriting. *arXiv preprint arXiv:2305.08272* (2023).

[5] Nicolas Bruno, Johnny Debrodt, Chujun Song, and Wei Zheng. 2022. Computation reuse via fusion in Amazon Athena. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1610–1620.

[6] Nicolas Bruno, César Galindo-Legaria, Milind Joshi, Esteban Calvo Vargas, Kabita Mahapatra, Sharon Ravindran, Guoheng Chen, Ernesto Cervantes Juárez, and Beysim Sezgin. 2024. Unified Query Optimization in the Fabric Data Warehouse. In *Companion of the 2024 International Conference on Management of Data*. 18–30.

[7] Nicolas Bruno and Rimma V Nehme. 2008. Configuration-parametric query optimization for physical design tuning. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 941–952.

[8] Bikash Chandra, Bhupesh Chawda, Shetal Shah, S Sudarshan, and Ankit Shah. 2013. Extending XData to kill SQL query mutants in the wild. In *Proceedings of the Sixth International Workshop on Testing Database Systems*. 1–6.

[9] 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).

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

[11] 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.

[12] Sriram Dharwada, Himanshu Devrani, Jayant Haritsa, and Harish Doraiswamy. 2026. LITHE: A Query Rewrite Advisor using LLMs. *Proceedings 29th International Conference on Extending Database Technology (EDBT 2026)* 29, 2 (2026), 233–246.

[13] Bailu Ding, Surajit Chaudhuri, Johannes Gehrke, and Vivek Narasayya. 2021. DSB: A Decision Support Benchmark for Workload-Driven and Traditional Database Systems. (2021).

[14] Bailu Ding, Vivek Narasayya, Surajit Chaudhuri, et al. 2024. Extensible query optimizers in practice. *Foundations and Trends® in Databases* 14, 3-4 (2024).

[15] 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.

[16] César Galindo-Legaria and Milind Joshi. 2001. Orthogonal Optimization of Subqueries and Aggregation. (2001), 571–581. doi:10.1145/375663.375748

[17] Cesar A. Galindo-Legaria, Torsten Grabs, Sreenivas Gukal, Steve Herbert, Aleksandras Surna, Shirley Wang, Wei Yu, Peter Zabback, and Shin Zhang. 2008. Optimizing Star Join Queries for Data Warehousing in Microsoft SQL Server. In *2008 IEEE 24th International Conference on Data Engineering*. 1190–1199. doi:10.1109/ICDE.2008.4497528

[18] 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.

[19] 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.

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

[21] Goetz Graefe and William McKenna. 1991. *The Volcano Optimizer Generator*. Technical Report. Colorado Univ at Boulder Dept of Computer Science.

[22] 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.]).

[23] IBM. 2025. Using EXPLAIN to determine UNION, INTERSECT, and EXCEPT activity and query rewrite. https://www.ibm.com/docs/en/db2-for-zos/13.0.0?topic=vntea-using-explain-determine-union-intersect-except-activity-query-rewrite

[24] Ralph Kimball and Kevin Strehlo. 1995. Why decision support fails and how to fix it. *ACM SIGMOD Record* 24, 3 (1995), 92–97.

[25] Jyoti Leeka and Kaushik Rajan. 2019. Incorporating super-operators in big-data query optimizers. *Proceedings of the VLDB Endowment* 13, 3 (2019), 348–361.

[26] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? (2015).

[27] 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).

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

[29] J Melton. 2003. ISO/IEC 9075-2: 2003 (E) information technology—database languages—SQL—Part 2: foundation (SQL/foundation).

[30] Microsoft. 2023. Hints (Transact-SQL) - Query. https://learn.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver16

[31] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (Seoul, Korea) *(VLDB '06)*. VLDB Endowment, 1049–1058.

[32] OpenAI. 2025. GPT-4o-mini. https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence

[33] OpenAI. 2025. GPT-5. https://openai.com/index/introducing-gpt-5

[34] Oracle. [n. d.]. SQL Plan Management in Oracle database. https://docs.oracle.com/en-us/iaas/database-management/doc/use-spm-manage-sql-execution-plans.html

[35] Partho Sarthi, Kaushik Rajan, Akash Lal, Abhishek Modi, Prakhar Jain, Mo Liu, Ashit Gosalia, and Saurabh Kalikar. 2020. Generalized {Sub-Query} Fusion for Eliminating Redundant {I/O} from {Big-Data} Queries. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 209–224.

[36] Tarique Siddiqui, Surajit Chaudhuri, and Vivek Narasayya. 2021. COMPARE: Accelerating groupwise comparison in relational databases for data analytics. *arXiv preprint arXiv:2107.11967* (2021).

[37] 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).

[38] 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).

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

[40] 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 *ICDE 2022*. 94–107.

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

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

[43] 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.

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

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

[46] Calisto Zuzarte, Hamid Pirahesh, Wenbin Ma, Qi Cheng, Linqi Liu, and Kwai Wong. 2003. Winmagic: Subquery elimination using window aggregation. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 652–656.