GRewriter: Practical Query Rewriting with Automatic Rule Set Expansion in GaussDB

Zhe Jiang[1] Zhaoguo Wang[1] Haoning Lan[1] Chuzhe Tang[1] Haoran Ding[1] Lefeng Wang[1]
Songyun Zou[1] Zhuoran Wei[1] Yongcun Liu[2] Xiang Yu[2] Yang Ren[2] Guoliang Li[3] Haibo Chen[1]
[1]{jz2000, zhaoguowang, sjtulhn, t.chuzhe, nhaorand, wanglefeng, zousy, 84461810, haibochen}@sjtu.edu.cn
[2]{liuyongcun, yuxiang44, renyang1}@huawei.com [3]liguoliang@tsinghua.edu.cn
[1]Institute of Parallel and Distributed Systems, Shanghai Jiao Tong University [2]Huawei Technologies Co.
[3]Department of Computer Science, Tsinghua University

ABSTRACT

Effectively rewriting a wide range of complex and diverse queries is critical for database systems. Huawei GaussDB has been experiencing limited extensibility of its existing query rewriter. The problem is rooted in the need for one-size-fits-all rewrites by its pipelined rewrite workflow and the source code-level coupling of rewrite logic. This makes it not only difficult to identify generic, broadly applicable rewrites but also engineering-intensive to program them into the system.

This paper presents GRewriter, GaussDB's new bolt-on extensible query rewriter powered by automated rewrite rule discovery. GRewriter sits atop the existing optimizer stack to explore useful rewrites, allowing a variety of rules to coexist and be selected on a per-query basis. A new rule language, G-DSL, is used to express rewrite rules so that the rewrite engine is not coupled with specific rules. To improve rewrite efficiency, a new rule index structure and a rewrite history cache are introduced. Rules in GRewriter are produced by an offline rule generator. With novel enumeration techniques and a new equivalence theorem, our rule generator can efficiently discover formally verified rules that are much more expressive than prior research prototypes. For operational convenience, GRewriter also supports manual rule authoring and interactive management of rules through familiar SQL interfaces.

GRewriter has been integrated into GaussDB and is gradually rolling out to customers. GRewriter equips GaussDB with over a hundred rules while maintaining negligible overhead (<1%). These new rewrite rules have enhanced query performance for two key customer applications, an ERP system and a Banking transaction system, reducing production query latency by up to 99.9%—from 26 seconds to just 17 milliseconds.

PVLDB Reference Format:
Zhe Jiang, Zhaoguo Wang, Haoning Lan, Chuzhe Tang, Haoran Ding, Lefeng Wang, Songyun Zou, Zhuoran Wei, Yongcun Liu, Xiang Yu, Yang Ren, Guoliang Li, and Haibo Chen. GRewriter: Practical Query Rewriting with Automatic Rule Set Expansion in GaussDB. PVLDB, 18(12): XXX-XXX, 2025.
doi:XX.XX/XXX.XX

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Query rewriter, which transforms a query into an equivalent yet more efficient form, is a key component in database systems that optimizes queries before execution. It relies on rewrite rules to transform queries into semantically equivalent forms with better performance [4, 13, 14, 19, 24, 25]. Therefore, the effectiveness of a query rewriter heavily depends on the quality and quantity of rules. Recent work has shown that adding new rules can bring up to 86% reduction in query latency of real-world applications [4].

Huawei GaussDB [18] is a commercial database system that holds a leading position in the Chinese market [17] and ranks among the top 10 database systems globally by share [12]. GaussDB takes a classic two-phase approach to query optimization, where first a rewriter optimizes the query at the logical plan level, then a planner generates a physical plan for execution. The rewriter transforms the logical plan parsed from the input query by a pipeline of rewrite functions, each performing a class of generic rewrite, such as predicate push-down. Rewrite conditions are hard-coded as various `if` statements, and the rewrite is done by directly modifying the internal data structure of the logical plan. The rewritten logical plan is then passed to the query planner to generate a physical plan for execution. This simple approach is straightforward to implement. Moreover, as rewrite logic is embedded in the source code, which gets compiled into binary, the rewrite process incurs low runtime overhead. Therefore, many other production systems also follow the same approach, such as MySQL [23] and PostgreSQL [16].

However, adding new rules to GaussDB's rewriter is challenging due to its tight coupling with the source code and the constraints of the pipelined workflow. To begin with, it requires a deep understanding of database internals and significant engineering effort to program such low-level rewrite functions. Furthermore, additional rewrite functions may unnecessarily lengthen overall latency, since they are always executed for all queries even if they only match an infrequent query pattern. The more complex and diverse the queries are, the more likely these issues will occur. As a result, GaussDB's rewriter has to be conservative in adding new rewrite functions, missing potential optimization opportunities.

To address the extensibility issue, an intuitive solution is to adopt a Cascades-style optimizer [13, 15]. Such an optimizer decouples complex rewrite logic from the source code and decomposes it into many basic rewrite rules. It relies on an iterative, heuristic-driven process to explore the rules and find useful combinations. Both logical rewrite and physical planning must be performed during the process so that costs estimated from physical plans can be

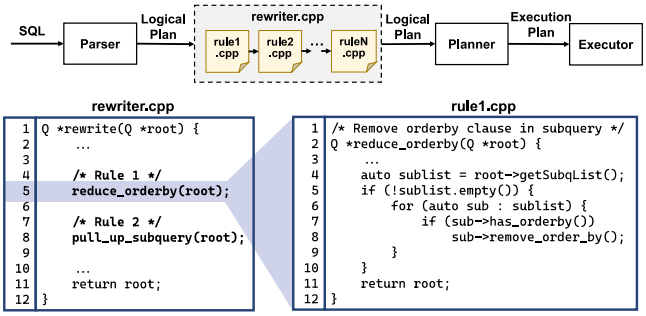

Figure 1: GaussDB's query rewrite process.

used as feedback to guide and prune the exploration. As a result, switching to a Cascades-style optimizer requires replacing not only the existing query rewriter but also the entire query planner.

For a production system like GaussDB, this is an extraordinarily expensive and risky undertaking, as it would involve discarding the current optimizer—comprising 642k lines of code—and developing a new one from scratch, all while ensuring it matches or exceeds the existing optimization capabilities. To the best of our knowledge, the only database system known to have undertaken such a transition is SQL Server. It is accomplished as part of the broader 7.0 revamp of its legacy Sybase engine written in C into a new C++ codebase [6].

This paper presents GRewriter, the new bolt-on, extensible query rewriter in GaussDB. Like Cascades, GRewriter employs decoupled rewrite rules and departs from the pipelined workflow. Unlike Cascades, GRewriter refrains from changing either the rewriter or planner, and instead works as a bolt-on component atop the existing optimizer of GaussDB. When GaussDB receives a query, GRewriter finds all applicable rewrite rules and applies them to obtain a set of equivalent logical plans. The original and rewritten plans are then passed to GaussDB's existing optimizer for further optimization and cost estimation. Only the plan with the lowest estimated cost is selected for execution. To improve rewrite efficiency, GRewriter employs a new index structure for fast rule matching and a rewrite history cache to amortize rewrite cost. Since GRewriter only performs logical rewrite, its bolt-on design should be easily adaptable to other relational database systems.

Instead of heuristically combining basic rewrite rules to produce useful transformation, GRewriter leverages recent advances in *automated rewrite rule discovery* [4, 11, 32] to directly generate more complex useful rewrite rules. Our rule generator is derived from WeTune, a state-of-the-art rule discovery research prototype [32]. It generates rules by first enumerating all possible candidates and then verifying their correctness and usefulness. While WeTune has successfully discovered missing rules in popular systems like Calcite [5] and SQL Server [22], it is still insufficient for commercial use due to its limited support for common SQL operators and crude rewrite conditions. Therefore, we have retrofitted three major enhancements to WeTune. First, a more expressive rule language, G-DSL, is proposed to specify rules involving more SQL operators and finer-grained rewrite conditions. Then, a set of pruning heuristics are developed to tame the exponential growth of the enumeration space due to the increased expressive power of G-DSL. Finally, a new verification theory is proposed that turns the verification of query plan templates into the verification of concrete SQL queries. As a result, a more powerful SQL equivalence verifier such as SQLSolver [10] can be integrated to verify the correctness of the generated rules.

GRewriter also supports manual rule authoring and interactive management of rule set at runtime. This allows resolving query performance issues in an agile and timely manner, without patching either the database system or the upper-layer applications. To ease writing complex rules, imperative constructs are incorporated into G-DSL. Administrators can use familiar SQL interfaces, such as `INSERT` and `DELETE`, to manage the rule set at runtime without disruption or downtime. Our experience shows that a slow query can be typically fixed within hours after being reported by customers. This is a significant improvement compared to the previous practice, where customers either had to wait for months-long feature release cycles of the database or manually patch their queries.

It took four months with 4 developers including part-time interns to develop an initial prototype, and later took another four months to productionize at GaussDB. This confirms that GRewriter is a low-cost solution. Now, GRewriter has been integrated into GaussDB and has been rolling out to its customers for evaluation for more than 3 months. Since then, GRewriter has generated 381 optimized rules applied to GaussDB, which only incur a 0.26% optimization overhead. These rules have been proven to enhance query performance for two key customers: an ERP system internally used at Huawei that covers 80% of its business volume, and a banking transaction system that is one of the biggest banking systems in China. GRewriter has brought significant improvement to production query latency, including a notable 99.9% reduction (from 26 seconds to 17 milliseconds) and a substantial improvement of nearly 2 minutes in other cases. Most of these newly discovered rules have not been implemented by popular relational databases such as PostgreSQL, MySQL, and SQL Server.

In summary, we make the following contributions:

- The design and implementation of GRewriter, the new bolt-on, extensible rule-based query rewriter in GaussDB that supports decoupled, dynamic management of rewrite rules.
- A new rule language, G-DSL, designed to express complex and diverse rewrite rules, suitable for both automated rule discovery and manual rule authoring.
- A new rule generator for finding rules written in G-DSL, with new enumeration techniques and a new equivalence theorem that enables efficient discovery of formally verified rules.
- The experience of serving production workloads with GRewriter-integrated GaussDB as well as detailed in-house evaluation, confirming its effectiveness and efficiency.

2 BACKGROUND AND MOTIVATION

2.1 Query Rewrite

Query rewrite is usually implemented as part of two-phase optimizer [25]. First, a query is subjected to a series of predefined rewrite functions by the query rewriter. Then, the query planner takes the rewritten query and generates a physical execution plan. Today's mainstream databases adopt similar optimizer architectures.

Take GaussDB as an example. Figure 1 shows its query rewriting process. Transformation of the logical plan is programmed as C++

Table 1: A counterintuitive fragment of a slow SQL from GaussDB's production workloads.

a. Original SQL Query	b. Rewritten SQL Query

`WHERE (  SELECT count(*)`
`  FROM t1 WHERE p1`
`) = (  SELECT count(*)`
`  FROM t1 WHERE p1 AND p2 )`
	`WHERE NOT EXISTS (`
	`  SELECT 1`
	`  FROM t1`
	`  WHERE p1 AND NOT p2 )`

functions. Both input and output are logical query plan trees. We present a simplified version of the rewriter's code. Rule1 eliminates redundant `ORDER BY` operators of subqueries. It retrieves the data structure representing subqueries and loops through each subquery to remove `ORDER BY` operators. The rewriter applies these functions in a predefined order, and the final output is fed to the query planner for physical planning. The rewriter in GaussDB is implemented in 7k LoC, incorporating only 22 rewrite functions. Other mainstream databases, such as PostgreSQL and MySQL, also adopt similar query rewriter architectures. Consequently, they all confront same issues discussed below.

Problem Statement. Such a rewriter architecture severely restricts the extensibility of its rule set. First, the rewriter's logic is embedded in the code, which makes it burdensome to add new rewrite rules or update existing ones. For example, the `OR-to-UNION` rewrite rule was initially missing in GaussDB's rewriter. It takes more than a half year to perform requirement review, design review, development, testing, and commercial trial to make the changes production-ready. This slow iteration process has been causing complaints from GaussDB customers.

Second, this architecture relies on one-size-fits-all rewrite rules that provide universal improvements for most queries when applied in a predefined order, making it difficult to optimize complex and diverse queries that are increasingly prevalent in cloud scenarios. Table 1.a shows a SQL statement from a GaussDB customer. Table 1.b is an ideal rewritten version that brings better performance because it reduces the number of table scans on `t1`. However, optimizing this pattern requires query plan structure matching and transformation logic written in C++. Adding one new rewrite function is not scalable and feasible for GaussDB since it is difficult to avoid rewrite quality regression when the rewrite pipeline consists of hundreds if not thousands of rules.

We envision GaussDB's query rewriter to be able to dynamically manage a rich set of rewrite rules and select the most effective one for each query, enabling more agile and efficient query optimization. Existing work has explored the bolt-on approach to extend rewrite capability with an additional rewrite layer and shows promising results in both low engineering effort and rewrite effectiveness [2–5, 26]. However, these rewriters still require non-trivial manual effort to populate rule sets. As a result, rule set scalability remains an unsolved challenge for them.

2.2 Automated Rewrite Rule Discovery

WeTune [32] is a state-of-the-art rewrite rule discovery framework. Its discovery process consists of three steps: rule enumeration, rule verification, and performance evaluation. A rule consists of a pair of source and target query plan templates for matching the original and rewritten queries, and a set of constraints that indicates the rewrite condition. To discover new rules, WeTune first enumerates all possible query plan templates with limited types of operators. It also restricts the size of templates to reduce the search space. Then, a Cartesian product of all templates with themselves forms template pairs. After that, constraints are enumerated for each template pair to form candidate rules. Finally, formal verification is performed to filter out incorrect rules and the remaining rules are evaluated with concrete queries and datasets to find the useful ones. Without any human efforts, WeTune discovers 35 useful rules, 22 missing in Calcite [5] and 7 missing in SQL Server [22].

Table 2: Generation time of 4-node rules using WeTune.

# of Op Types	# of Plan Templates	# of Candidate Rules	Time to Terminate
6	4800	9.4×10^{22}	7 days
12	116,640	2.1×10^{28}	4284 years

However, WeTune's techniques are insufficient for production use. Specifically, WeTune has limited support for common SQL operators and constraints. For instance, rules shown in Table 1 cannot be generated by WeTune because 1) the `count()` aggregate function, `EXISTS` operator, and scalar subqueries are not supported, and 2) the `WHERE` predicate is treated as a black box, making it impossible to specify constraints involving its internal structures.

Making WeTune practical is non-trivial. First, the enumeration space grows exponentially with the number of operators and constraint types. As a result, it has to be restricted to the following basic operator types: projection, selection, in-subquery, join, and input, and simple constraints. As shown in Table 2, if we simply extend the operator types to 12, there will be a more than 200M-fold increase in the number of candidate rules, leading to a prohibitive rule generation time. Second, WeTune depends on the verifier's ability to check the correctness of the enumerated rules. But, the built-in verifier used by WeTune can only handle a limited range of SQL features and simple rule formats [32]. Thus, even if more complex rules are enumerated, they cannot be formally verified. Although many advanced SQL verifiers have emerged recently, they focus on concrete queries rather than rewrite rules [8, 10, 31, 33].

2.3 Goal and Challenges

We aim to equip GaussDB with a new bolt-on extensible query rewriter with automatically discovered rewrite rules to enable GaussDB to handle a broader range of complex and diverse real-world queries, while minimizing engineering effort. Realizing this vision, however, faces three challenges.

The first challenge lies in the representation of rules. There are three requirements. First, it should allow a clean separation of the rule's structure and semantics from the rewriter's implementation. Second, it should be expressive enough to capture a wide range of rewrite patterns. Third, it should be friendly to both enumeration-based automated discovery and manual rule authoring by experts. However, existing rule representations do not simultaneously meet all these requirements [4, 5, 27, 32].

The second challenge is the efficiency of the rule-based rewriter. The bolt-on rewriter works as an online component in the critical

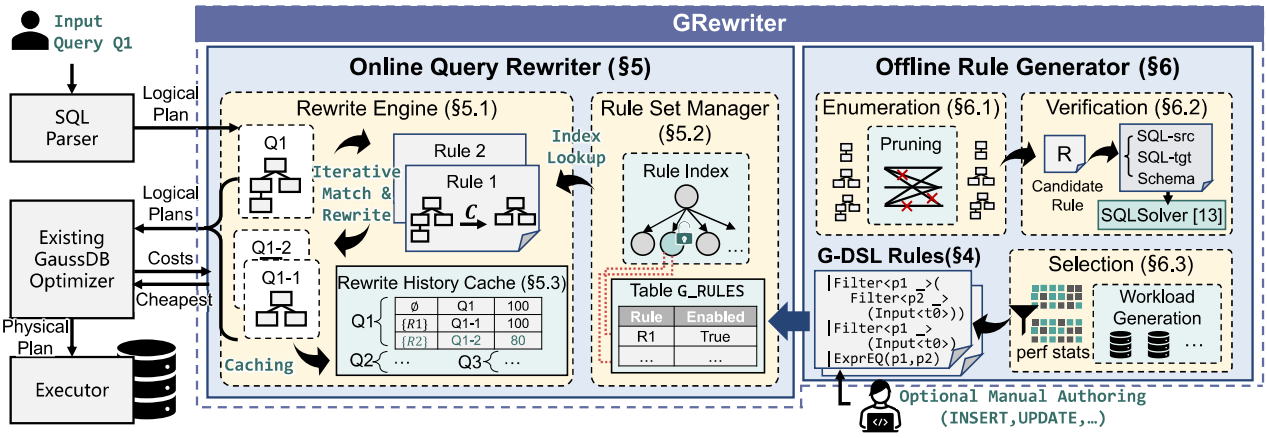

Figure 2: GRewriter architecture.

path of query processing, making it critical to minimize its impact on the overall latency. However, with a larger set of rewrite rules and the fact that different queries may be optimized by different rules, the rewriter needs to explore many potential rewrites. Therefore, how to ensure the search efficiency and balance between the exploration and processing cost is a challenge.

The final challenge lies in the capability of the rewrite rule generator. To find useful rules, we need a more powerful generator than WeTune that can not only scale to more operators and constraints, but also be able to verify the correctness of the rules it generates. However, it is challenging to handle the exponential growth of enumeration space and to formally verify the more expressive rules.

3 GREWRITER OVERVIEW

Figure 2 shows the architecture of GRewriter, the new bolt-on, extensible rule-based query rewriter in GaussDB powered by automated rewrite rule discovery. It consists of an *online rewriter* and an *offline rule generator*, bridged by a new rule language, *G-DSL*.

G-DSL (Section 4) is an expressive language that supports a wide range of SQL operators and fine-grained constraints. It supports 12 common SQL operators and incorporates the idea of *expression destructuring* to allow a more flexible specification of rules. Rules written in G-DSL are decoupled from how the rewriter matches and applies rules and is suitable for both programmatic enumeration and manual authoring.

The online rewriter (Section 5) takes in a user query as a logical plan and iteratively explores possible rewrites, producing multiple rewritten logical plans. It calls into GaussDB's existing optimizer to estimate the costs of the original and rewritten plans. The plan with the lowest estimated cost is selected for execution. To improve efficiency, a new rule index structure and a rewrite history cache are introduced, and they are carefully designed to work in the presence of dynamic rule set updates.

The offline rule generator (Section 6) follows the enumeration-based paradigm of automated rewrite rule discovery. To enumerate G-DSL rules, new procedures and heuristics are introduced to make the enumeration space manageable. To verify now more expressive rules, a new verification theorem is developed to turn the verification of query plan templates into the verification of concrete SQL queries, making it possible to integrate with a more powerful SQL equivalence verifier, SQLSolver [10]. The generator evaluates verified rules using multiple synthetic datasets, and only those consistently bring at least 10% latency reduction are kept.

4 REPRESENTATION OF RULES

G-DSL is our language to specify rules. A rule consists of three parts: a source plan template, a target plan template, and constraints. When the logical plan of a query matches the source plan template and the constraints are satisfied, it can be rewritten to another one that matches the target plan template.

4.1 Plan Template

Plan templates are tree-structured: each node represents a relational operator and each edge represents the relations passed between operators. Formally, a plan template is defined as `op<params>(ops)`. `op` is the operator of the root node, `params` are the parameters of `op`, and `ops` are the children of the node. A plan template matches a logical plan subtree only when they have the same tree structure. Each leaf node is always an `Input` operator. Its parameter `r` is a symbolic wildcard relation, which can be instantiated to arbitrary logical plan subtrees, e.g., a single concrete table or a subquery.

Operator parameters are symbolic, which are placeholders for the concrete values in the query. There are three types of symbolic parameters: symbolic attribute lists, symbolic expression lists, and symbolic relations. The comma between two symbolic lists represents the concatenation of all elements in two lists. For example, projection is denoted as `Proj<e(a),s>`, where `a` is a symbolic attribute list, `e` is a symbolic expression list, and `s` is the symbolic relation produced by the projection. `e(a)` returns expressions built from `a`. As a syntax sugar, the underscore symbol "_" is used for matching parameters whose specific values are irrelevant to the rewrite process, i.e., not used by the constraints, including the implicit equivalence relationships implied by the repetitive use of the same symbol.

Table 3: Operators supported in G-DSL.

Operator Type	Definition	Matching Query Fragment
Input	`Input<r>`	`... r ...`
Projection	`Proj<e(a),s>`, `ProjDedup<e(a),s>`	`(SELECT [DISTINCT] e(a) FROM ... ) AS s`
Selection	`Filter<e(a)>`	`... WHERE e(a)`
Join (left & inner)	`JoinLeft<a0,a1>`, `JoinInner<a0,a1>`	`... LEFT/INNER JOIN ... ON a0 = a1`
In-Subquery	`InSub<e(a)>`	`... WHERE e(a) in ( <subquery> )`
Aggregation	`AggFunc<e0(a0), e1(a1), e2(a2)>`	`SELECT e0(a0), Func(e1(a1))...GROUP BY e0(a0) HAVING e2(a2)`
Union, Union-All	`Union<>`, `UnionAll<>`	`... UNION [ALL] ...`
Exists-Subquery	`Exists<>`	`... WHERE EXISTS ( <subquery> )`
Limit	`Limit<a0, a1>`	`... LIMIT a0 OFFSET a1`
Sort	`SortAsc<e(a)>`, `SortDesc<e(a)>`	`... ORDER BY e(a) ASC/DESC`

Table 3 defines the supported operators in G-DSL, accompanied by illustrative SQL fragments. The projection operator can match fragment `(SELECT c0+c1 FROM ...) AS t` with `a` corresponding to the concrete attribute list `[c0,c1]`, `e(a)` corresponding to `[c0+c1]`, and `s` corresponding to the alias name `t`. Aggregate functions are denoted as `AggFunc<e0(a0),e1(a1),e2(a2)>`, where `Func` is the function name (e.g., `sum`). In this operator, `e0(a0)` specifies the expressions in the `GROUP BY` list, `e1(a1)` represents the input to the aggregate function, and `e2(a2)` represents the `HAVING` condition. For example, `SELECT c0, min(c1) FROM ... GROUP BY c0 HAVING c2>0` can be represented by an `AggMin` node: `a0`, `a1`, and `a2` correspond to `[c0]`, `[c1]`, and `[c2]`, respectively. `e0(a0)`, `e1(a1)`, and `e2(a2)` correspond to `[c0]`, `[c1]`, and `[c2 > 0]`, respectively. Other operators are defined similarly and we omit due to space limitations.

4.2 G-Constraints

Constraints are expressed as G-Constraint statements, each returning a boolean value. Given a query whose logical plan matches the source plan template of a rule, constraint statements are evaluated one by one. Only when all statements return true is the rule applicable. Statements involving the target template's symbols are always considered true, as they are used to determine the values of rewritten queries. There are five types of statements. The former three are declarative and the latter two are imperative.

Expression Destructure. `EXP(S1,S2,...):=S0` decomposes symbol `S0` into symbols `S1`, `S2`, etc. It returns true if the structure of the query fragment corresponding to `S0` matches the syntax structure specified by `EXP`. Otherwise, it returns false. `EXP` can be either arithmetic or boolean operators (e.g., `AND`, `>`, and `+`). For example, given a symbol `e` that corresponds to fragment `a>1 AND a<2`, `AND(e1,e2):=e` will return true. `e1` and `e2` are assigned to be `[a>1]` and `[a<2]`, respectively. If symbols on the left side are already defined, this statement returns false if the decomposed fragments do not match the defined symbols. When `S0` is a list of expressions or attributes, scalar expressions indicated by `EXP` are promoted to element-wise list operations.

Symbol Definition. Statement `S0:=EXP(S1,S2,...)` assigns a symbol *S0* to `EXP(S1,S2,...)` which combines the symbols *S1*, *S2*, and others into a new syntax structure. This statement can also redefine an existing symbol. Since it does not check constraints, it always returns true.

Constraint Check. Statement `[!]CONS(S0,S1,...)` checks the relationship among symbols and integrity constraints. Specifically, `AttrEq(a0,a1)`, `TableEq(r0,r1)`, `ExpressionEq(e0,e1)` checks if two attribute lists, tables, and expression lists are the same, respectively. `AttrsSub(a,r)` checks if an attribute list `a` belongs to a relation `r`. `Notnull(a,r)` checks if the attribute list `a` in the relation `r` is not null. `UNIQUE(a,r)` checks if the attribute list `a` in the relation `r` is unique. `FOREIGN (a0,r0,a1,r1)` checks if the attribute `a0` in the relation `r0` is a foreign key that references `a1` in `r1`. We also introduce the negation operator `!`, which means the constraint should not be satisfied.

Conditional. For the statement `IF ST1 DO ST2;... ELSE ST3;... END IF`, if `ST1` returns true, `ST2;...` will be executed and the conditional statement returns true if and only if all statements in `ST2;...` return true. Otherwise, `ST3;...` will be executed and the conditional statement returns true if and only if all statements in `ST3;...` return true.

Loop. For the statement `WHILE ST1 DO ST2;... END WHILE`, when `ST1` returns false, the loop terminates and the loop statement returns true. If any statement in `ST2;ST3;...` returns false, the loop also terminates and returns false. Loops are usually used to recursively decompose the query fragment into sub-fragments and check their syntax structures without a presumed number of iterations.

4.3 Example and Discussion

Example. Figure 3 gives an example rule written in G-DSL and shows the whole match and rewrite process of G-DSL. If a query contains a Filter operator whose parameters are two predicates connected by an OR operator, it can be rewritten as the union of the results filtered by each predicate separately. Given a query, we first obtain its logical plan tree and try to match each subtree against the source template. If a match is successful, we record the matched variable values in a variable mapping. We then execute the G-Constraint to check constraints via the variable mapping. Specifically, the G-Constraint checks whether the Filter operator consists of two predicates connected by an OR operator. Finally, we construct the new logical plan based on the variable mapping, the target template, and the G-Constraint.

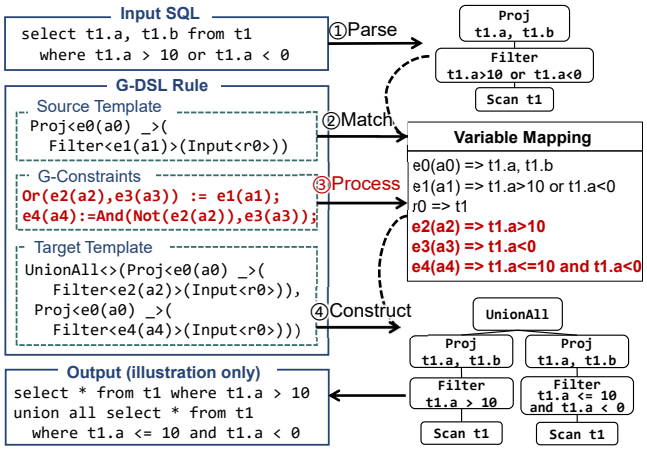

Figure 3: An example rule written in G-DSL. This rule transforms OR predicates into UNION ALL.

Other Rule Representations. SlabCity [11] formally defines the syntax to describe supported queries. However, it cannot be used to define rewrite rules as it lacks symbolic parameters and constraints, which are essential for describing the condition for a rule to apply. Meanwhile, although there are some other rule representations, they cannot simultaneously meet all three requirements in Section 2.3. The DSL in WeTune [32] cannot support some SQL features and constraints supported by G-DSL. In evaluation, 381 discovered useful rules cannot be expressed in WeTune's DSL. PostgreSQL allows users to define custom data transformation and view creation for specific tables or columns. It is hard to create general rules applicable for various workloads. The DSL in QueryBooster [4] is more expressive than WeTune's DSL but cannot support constraints in imperative constructs, which are necessary to express some useful rules. In evaluation, rules with such constraints can bring up to 72% latency reduction for production queries. To enhance the expressiveness, CockroachDB [27] represents rules with a language called OptGen, which expresses imperative constraints by Go procedures. Apache Calcite [5] defines rewrite rules as individual Java classes, which also supports imperative constraints. However, rules in these representations would be intractable to enumerate and verify, which essentially requires the ability to enumerate and verify arbitrary programs written in Go or Java. We carefully design the syntax of G-DSL such that it is more friendly to enumeration-based rule discovery while being expressive enough to capture a wide range of rewrite patterns.

5 ONLINE REWRITER

The rewriter is responsible for exploring potential rewrites and outputting an optimized logical query plan for subsequent execution. At a high level, it takes in a logical query plan parsed from the original query, performs iterative rewrites, and produces a set of semantically equivalent, potentially more efficient logical plans. Among all equivalent candidates, the one with the lowest cost estimated by GaussDB's existing optimizer is selected for execution.

5.1 Workflow

Algorithm 1 shows the general workflow of the online rewriter. For simplicity, we skip rewrite caching (lines 2 and 20) for now, and discuss it in Section 5.3. First, given a logical plan Q, the rewriter prepares an empty *rewrite path* without applying any rule (line 4). A rewrite path is a tuple that consists of the starting and ending plans, the rules applied, the estimated cost, and whether the iterative exploration should continue. It then starts a loop over the paths marked unexplored (lines 7–19). For each unexplored path, it retrieves all potentially applicable rules from the rule set (line 8). This lookup only performs a structural match between the given logical plan and the source template of each rule. The lookup algorithm is discussed in Section 5.2. Next, for each potential match, the rewriter consults the system catalog for the relevant schema information and discard rules whose rewrite conditions are not satisfied (line 9). It then applies each remaining rule to the logical plan, generating a set of new paths with rewritten, equivalent plans and the corresponding rules applied (lines 10–13). Now all the possible rewrites on the current path are explored, so we mark it explored with `True` (line 14). This process is repeated using previously rewritten paths as the starting point until all paths is marked explored (i.e., no more rules can be applied) or a fixed number of iterations is reached (line 16). Finally, the rewriter passes all generated plans along with the initial plan to the existing optimizer for physical planning and cost estimation (lines 17–19). The cheapest plan is returned for final execution (lines 21–22).

5.2 Storing and Indexing Rules

GRewriter stores rewrite rules in a separate system table, `G_RULES`, which persists the rules in plain text and remembers if a rule is enabled or not (Section 5.4). To speed up rule lookup and matching, GRewriter introduces a new index structure, called the Rule Index, to efficiently store and retrieve rules. Rule Index is a concurrent trie structure that maps the *fingerprint* of a plan template to the row IDs of rules whose source templates have the same fingerprint. A fingerprint of a tree structure is defined as a string generated through a preorder traversal of the template tree, retaining only node type information. For example, the fingerprint for a query plan structure `Filter(Filter(Input))` is represented as `FFI`. The Rule Index maps the fingerprints of rule templates to their corresponding rule sets. When a query arrives, the rewriter computes its fingerprint and queries the Rule Index. When searching in the trie, all rule sets associated with the nodes along the path of the fingerprint are collected and returned. For instance, when querying `FFI`, the process retrieves the rule set at the `F` node, followed by the set at the `FF` node (treating the `I` node as a wildcard). Additionally, all suffixes of the fingerprint are queried. For `FFI`, this includes `FFI` and `FI`. This design avoids exhaustive rule set searches, significantly enhancing rewrite performance, particularly for large rule sets. To ensure correctness under concurrent access, the Rule Index employs a hierarchical read-write locking mechanism.

5.3 Rewrite History Caching

Caching helps to avoid redundant rewriting and to chunk the long iterative rewriting process for queries with many applicable rules. Each time a query plan arrives, the rewriter first checks the rewrite

Algorithm 1 Rewrite Workflow

```
1:  function REWRITE(Q)
2:      paths ← cache.get(Q)
3:      if paths.empty() then
4:          paths.add(Path(Q, Q, {}, F))
5:      iter ← 0
6:      repeat
7:          for all path ∈ paths.getUnexplored() do
8:              psblRules ← RULESET.get(path.Q)
9:              rules ← consCheck(psblRules, path.targetQ)
10:             for all rule ∈ rules do
11:                 newQ ← apply(rule, path.targetQ)
12:                 usedRules ← path.usedRules + rule
13:                 paths.add(Path(path.srcQ, newQ, usedRules, F))
14:             path.explored ← True
15:         iter ← iter + 1
16:     until !paths.hasUnexplored() or iter == 3
17:     for all path ∈ paths do
18:         if path.plan == null then
19:             path.plan ← native.plan(path.targetQ)
20:     cache.put(Q, paths)
21:     optimalPlan ← paths.getMinCostPlan()
22:     return optimalPlan
```

history cache (line 2). The rewriter will start the iterative rewriting from scratch if we have not cached any path for the input query plan. It may also continue from cached unexplored paths that are created in previous rewrite invocations where the exploration was not finished but the iteration limit was reached. Otherwise, if all cached paths are marked explored, the cached plan with the lowest average estimated cost is returned and iterative rewriting is finished. Finally, when the existing optimizer has returned the estimated cost of each plan, the rewriter updates the rewrite history cache with the new written plans and their costs (line 20).

We implement the rewrite history cache as an in-memory concurrent multimap. It uses parameterized query strings, with all parameter values removed, as the keys, and maps them to a list of previously explored rewrite paths along with their most recent five estimated costs. Remembering multiple estimated costs helps avoid trashing caused by queries that only differ in parameter values but have different optimal plans. Since the cost of cache entries may become stale, we implemented a trigger to clear the cost information in the Rewrite history Cache whenever the existing optimizer updates the statistical data used in its cost model. This can happen when the underlying data referenced by a query is modified, or when the cost model performs its periodical update.

5.4 Dynamic Rule Set Update

Administrators can easily manage rules by modifying the `G_RULES` table using familiar SQL syntax. For example, a statement like `UPDATE G_RULES SET ENABLE=OFF WHERE RULE_ID=?` to deactivate a specific rule or use `DELETE/INSERT` to remove/add a new rule. When rules are added or deactivated, the Rule Index is updated to reflect the changes. Our concurrent index uses fine-grained locking to avoid unnecessary blocking between concurrent readers and writers. Read requests to the index sequentially acquire read locks on each node along the path. When inserting a new rule, read locks are acquired sequentially on each node along the path, culminating in a write lock on the node where the rule is to be inserted. If the corresponding node cannot be found, meaning no existing rule in the index matches the source template of the rule to be inserted, a new node must be created in the index. Thus, a write lock on the parent node must be acquired beforehand. Rule set updates also trigger a rewrite history cache partial invalidation. Specifically, when we add a rule with a certain source template, we need to traverse all paths in the cache to find query plans with the same pattern and mark their explored status as false. This ensures the query will have the opportunity to be optimized by new rules upon its next encounter. Similarly, when a rule is removed or deactivated, corresponding cached paths are removed.

6 OFFLINE RULE GENERATOR

The generator follows the three-step workflow of prior work [32], i.e., enumeration, verification, and selection, while adapting each step to accommodate the more expressive G-DSL rule language.

6.1 Rule Enumeration and Pruning

The rule enumerator is responsible for generating a large number of candidate rules written in G-DSL. First, we enumerate all possible templates and combine them into template pairs. This process is similar to the work mentioned in Section 2.2, but we use the extended operator types defined in Table 3. Then we enumerate constraints for each template pair to form a candidate rule. We extract all expression symbols from the source and target templates and combine them to form possible G-Constraint statements, including expression destructure, symbol definition and constraint check. With the new operator types and constraints introduced, we propose the following techniques to narrow the search space.

Template Pruning. After enumerating plan templates, we can prune templates that lead to invalid or unpromising rules. Templates that do not adhere to correct SQL syntax are considered invalid. A template like `Join(Filter,*)` falls into this category, since when a SQL is parsed into logical plan, `Filter` node cannot directly appear under a `Join` node without `Proj`. Note that, although a simple fix could sometimes turn these templates into valid ones, we choose not to fix them since the correct ones should already be enumerated in the first place. To prune unpromising templates, we collect queries from user workloads. Source templates that do not appear in the query set are considered unpromising and discarded. In other words, we only enumerate the source templates that have appeared in the collected query set.

Template Pair Pruning. After we have all candidate templates, we need to enumerate all possible source-target pairs before adding constraints. A set of heuristics are used to prune the template pairs. One notable heuristic is built on the potential transitivity of the inequality property. Suppose there are three templates t_a, t_b, and t_c and we have already paired t_a and t_b with all possible constraints. If our verifier, which will be introduced shortly, finds that t_a and t_b are never equivalent under any constraints, but there are constraints under which t_a could be equivalent to t_c, then we can consider t_b

and t_c to be likely never equivalent under any constraints. This heuristic allows us to skip a large number of template pairs like t_b and t_c, significantly reducing the enumeration space.

Candidate Rule Pruning. Given a pair of source and target templates, we need to enumerate all possible constraints. We enumerate all possible constraint statements as defined in Section 4.2 and each subset of constraints is paired to the source and target templates to form a candidate rule. We prune candidate rules by checking the containment relationship between the constraint sets and known verification results. Specifically, if a candidate rule with a constraint set t_a is verified to be correct, then all candidate rules with a constraint set t_b that is a superset of t_a are also correct, and we can skip verifying them. Likewise, if a candidate rule with a constraint set t_a is verified to be incorrect, then all candidate rules with a constraint set t_b that is a subset of t_a are also incorrect, and we can skip verifying them as well.

6.2 Rule Verification

GRewriter develops a rule verifier for the enumerate rules that fully reuses the verification capabilities of existing query verifiers Query verifiers [9, 10, 31, 33] are powerful but they target concrete queries rather than rules. To fully reuse their verification capabilities, we observe that given a rewrite rule, there are finite representative concrete query pairs such that their equivalence implies rule correctness. For example, assume the source template is `Proj<a0,r2> (Proj<a0, r1>(Input<r0>))`, the target template is `Proj<a0, r1>(Input<r0>)`, and the constraint is `AttrsSub(a0,r0)`. If `SELECT C0 FROM (SELECT C0 FROM R0)` and `SELECT C0 FROM R0` are equivalent, the rule is correct. The schema is `R0(C0 INT)`. Based on the observation, we verify rules by generating representative query pairs. First, we assign each symbolic expression in templates to a random function. Second, for each symbolic schema, we enumerate all possible concrete schemas under two cases: the schema only contains the attributes used in the query, or the schema contains one more unused attribute. Since symbolic schemas are finite, enumerated concrete schemas are also finite. Finally, since each operator in G-DSL has corresponding SQL features, it is easy to translate the plan templates into concrete queries that can be verified by existing verifiers (e.g., SQLSolver [10]).

The correctness of our verification algorithm is ensured by Theorem 6.1. Due to space limitation, we only provide a proof sketch here. The detailed proof can be found in our appendix [1].

Theorem 6.1. *For each candidate rewrite rule, if each representative pair of queries is equivalent, the rule is correct.*

Proof Sketch. We prove the theorem by proving that for each non-representative query pair p_1, there exists a representative query pair whose equivalence implies the equivalence of p_1. We classify p_1 into three cases. First, p_1 can become the same as a representative query pair p_2 by renaming p_1's attributes and tables. Specifically, there is a one-to-one name mapping between names used in p_1 and p_2. Since renaming does not affect whether two queries are equivalent, the equivalence of p_2 implies the equivalence of p_1. Second, after renaming, p_1 is the same as a representative query pair p_2 except for concrete expressions. The expressions of p_2 are uninterpreted user-defined functions, which can represent any kind of concrete expression. Thus, the equivalence of p_2 implies the equivalence of p_1. Third, after renaming, p_1 is the same as a representative query pair p_2 except for concrete schemas. According to the steps of constructing representative query pairs, their schemas will differ in attributes not used in queries or the length of some attribute lists. The attributes not used in queries do not affect the equivalence of queries. For the attribute lists, existing verifiers treat them to be a whole and the length will not affect the verification results. Thus, the equivalence of p_2 implies the equivalence of p_1. Note that p_1 always belongs to one of the above cases, because queries in p_1 and representative query pairs all satisfy the syntax structures and constraints specified in the rule. They can only differ in concrete names, expressions, and schemas. In conclusion, the equivalence of representaive query pairs can imply the equivalence of any non-representative query pairs, which further implies the rule correctness. Theorem 6.1 is proved to be true. □

6.3 Rule Selection

The rules obtained through enumeration and verification ensure semantic equivalence of SQL before and after rewriting, but whether the rewrite improves SQL performance is unknown. We developed a workload generation algorithm for G-DSL rules. Given a schema, the algorithm generates an SQL query that can be matched by the rule based on the rule's source template and constraints. It is similar to transforming rules into SQL pairs described in Section 6.2, but instead of enumerating all possible schemas, it generates only one SQL based on the existing schema. Once the SQL is generated, we compare the performance of the rewritten and original SQL under different data sizes for that schema. We have prepared six setups for data distribution and data size: micro benchmarks in uniform distribution with 100k and 1M rows, the official TPC-H [30] 1GB and 5GB datasets, and the official TPC-DS [29] 1GB and 5GB datasets. Only rules that bring at least 10% latency reduction in all setups are considered useful and incorporated into the rewriter.

7 DISCUSSION

Exponential Growth of Paths. While exponential growth of paths explored is possible due to the recursive nature of our rewrite process (Algorithm 1), three design choices help reduce its likelihood and performance impact. First, as mentioned in Section 6.1, we prune unpromising source templates, keeping only those that appear in the queries collected from application workloads. Therefore, rewritten queries are unlikely to be recursively matched. Second, due to our manual cap on the number of rewrite iterations, which is currently set to three (line 16 in Algorithm 1), the number of paths explored during a single query execution is limited. This number is at most 6 in our evaluation using GaussDB customer workloads. Finally, offline filtering that removes rules not bringing actual latency reduction (Section 6.3) reduces the number of applicable rules, which also helps lower the number of paths.

Rule Prioritization. Since we limit the number of iterations during rewrite, when there are multiple applicable rules, rule prioritization helps producing better plans more quickly, i.e., in fewer repetition of queries, which is especially useful for infrequent queries. However, prioritizing rules opens up a large design space and can be a challenging task. For example, LearnedRewrite [34] leverages Monte Carlo Tree Search (MCTS) and GenRewrite [20] utilizes

LLMs to select appropriate rule sequence for the incoming query. Meanwhile, heuristics such as the historical performance gains of rules may also be helpful in prioritizing rules. Our current implementation simply selects rules based on their inclusion times, and we leave rule prioritization as future work.

Adapting to Specific Workloads. Different workloads may benefit from different rewrite rules. Currently, GRewriter exploits this observation by collecting workload queries to extract source templates for offline rule generation and provides APIs for administrators to interactively manage the rules. Further adapting to dynamically changing workloads may require continuously monitoring the workload and automatically adjusting the active rule set. For example, we can sample queries and periodically trigger the rule generation process to produce new rules. Alternatively, for alternating workload patterns like workday-weekend cycles, selective activation of rules based on their recent performance gains may be beneficial. Exploring these options is left as future work.

8 EVALUATION

We evaluate GRewriter to answer these questions:

(1) How does GRewriter benefit GaussDB customers?
(2) What is the quantity and quality of rules added by GRewriter?
(3) How is the efficiency of components in GRewriter?

8.1 Setup

Implementation. We implemented the online rewriter from scratch using 26k lines of C++ code, which has been successfully integrated into the current preview version of GaussDB for beta evaluation by customers. It includes configurable options to determine whether to disable the Rule Index and Rewrite History Cache. The offline rule generator is based on the source code of WeTune[32] and SQLSolver[10]. Besides, we wrote 4k lines of Java code for missing operators and expressions, 15k lines for our new G-Constraint constraints enumeration and search space pruning.

Testbed. Production workloads are evaluated using on-premise deployment of the latest GaussDB. In-house benchmarks are evaluated on a cloud server with 8 vCPUs and 32GB DRAM. Our offline rule generator runs on a server with Intel Xeon Gold 5317 CPU (3.00GHz, 24x2 cores), 188GB DRAM, and 7TB NVMe SSD.

Workloads. There are two types of workloads used to evaluate GaussDB with GRewriter. The first is the production workload which consists of two customer applications deployed on GaussDB: 1) a banking transaction system from a top domestic bank in China, serving tens of millions of corporate customers and much more personal customers. This banking transaction system is one of the largest banking systems in China; 2) an ERP system internally used at Huawei, which has been used to serve all Huawei's business and cover 80% of its business volume. GaussDB is the core database system backing these applications handling a total of 1250 GB data and a high volume of queries. We report the performance benefits brought by GRewriter on their production queries[1].

The second is the in-house workload with six datasets for detailed performance analysis. Two datasets are created from a uniform distribution, containing 100k and 1M rows, respectively. They share the same schema designed for matching all generated rules. There are four tables and each table has six integer columns and one string column[2]. The other four datasets are created from the standard schemas of the TPC-H and TPC-DS benchmarks, each populated with 1GB and 5GB data using their official data generation tools. To discover the useful rules, we synthesize queries that can be rewritten by the rules and evaluate across the six datasets to check if the rewriting will incur performance improvement. For the Rule Index and Rewrite History Cache evaluation, we use the standard queries from TPC-H [30] to evaluate their performance impacts.

8.2 Benefits to Customer Applications

We have partnered with teams from these two customers to evaluate GRewriter on-premise with their production workloads. We focus on critical components in their applications where GRewriter is most likely to bring benefits, i.e., those that not only are business-essential but also face high query volume or latency requirements. These queries span from OLTP-style reads and updates, like management of operational routing for assembly items, to OLAP-style analytics, like producing portfolio reports for financial products.

Query Speed-Up. We first report the end-to-end improvement on query performance in GaussDB production workloads. After enabling GRewriter with newly generated rules, we have observed speed-ups over 18 recurring queries. The absolute query latency varies greatly as they cover a wide range of workloads, and we therefore show the relative speed-up ratio for each query in Table 4. We observe that GRewriter brings a maximum speed up of 1,543×, and more than half of the queries have a 3.5× speed up or more. In terms of absolute time saved, eight queries have saved more than 500 milliseconds, and the maximum time saved nearly reaches 2 minutes. Interestingly, while some query only have a seemingly moderate relative speed-up, such as Q1 and Q2, the absolute time saved is significant as their latency is initially high.

How Rewrite Helps. We next examine the rules that successfully optimized these production queries. As summarized in Table 5, these rewrites are resulted from 7 different new rules that were absent in the existing rewriter in GaussDB. Among them, 5 were discovered via enumeration and 2 were manually crafted by experts experts from GaussDB. We observe that, these rules are useful in optimizing a wide range of queries and bring notable speed-ups.

Q3 shows the largest speed-up, from 26 seconds to 17 milliseconds, by transforming a `COUNT=0` predicate to a join operation. This query has a complex `WHERE` predicate that checks the return value of a scalar subquery: `... FROM t1 WHERE (SELECT COUNT(*) FROM t2 WHERE t1.c=t2.c) = 0`. Executing this query results in nested execution of the subquery for each row in `t1`, leading to a high latency. This is effectively an anti-join operation and there exists an efficient anti-join physical operator in the GaussDB optimizer. Rewriting the query into `... FROM t1 LEFT JOIN t2 ON t1.c=t2.c WHERE t2.c IS NULL` allows the planner to use it. However, this rewrite does not fall into any of the existing 22 functions in the GaussDB rewriter. Fortunately, GRewriter is able to automatically discover and apply this rewrite.

[1] Due to the confidentiality of the data, we do not disclose the specific queries.

[2] Our rules are oblivious to data types. The use of a string column is simply for showcasing GRewriter's ability to handle different data types.

Table 4: End-to-end latency of production queries, before and after optimization.

Query	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9
Original	4.7 min	4.3 min	26 sec	6.5 sec	791 ms	690 ms	623 ms	620 ms	608 ms
Optimized	2.8 min	2.5 min	17 ms	5.4 sec	243 ms	190 ms	147 ms	173 ms	118 ms
Speed-Up ↑	1.7×	1.8×	1543×	1.2×	3.3×	3.6×	4.2×	3.6×	5.2×
Query	**Q10**	**Q11**	**Q12**	**Q13**	**Q14**	**Q15**	**Q16**	**Q17**	**Q18**
Original	602 ms	602 ms	127 ms	52 ms	50 ms	48 ms	47 ms	15 ms	13 ms
Optimized	96 ms	96 ms	117 ms	15 ms	14 ms	14 ms	13 ms	14 ms	11 ms
Speed-Up ↑	6.3×	6.3×	1.1×	3.5×	3.6×	3.4×	3.6×	1.1×	1.2×

Table 5: Rules used by GRewriter, the geometric means of speed-up and whether listed DBs performed the same rewrite.

Rule	Queries	GaussDB+GRewriter (Speed-Up↑)	PostgreSQL	MySQL	SQL Server
Transform `COUNT=COUNT` to `EXISTS`.	Q13, Q14, Q15, Q16	✓ (3.5×)	-	-	-
Remove same condition joins.	Q9, Q10, Q11	✓ (5.9×)	-	-	-
Remove unused joins.	Q7, Q8, Q12	✓ (2.6×)	-	-	-
Predicate pull-up to scalar subqueries.	Q5, Q6, Q18	✓ (2.4×)	-	-	-
Build common table expressions.	Q1, Q2, Q4, Q17	✓ (1.4×)	-	✓	✓
Transform subquery counting to joins.	Q3	✓(1543×)	-	-	✓
Filter and join condition propagation.	Q5, Q6	✓ (3.4×)	-	-	-

Another typical rewrite pattern is the 1st rule in Table 5, which has successfully optimized Q13, Q14, Q15, and Q16 with an average speed-up of 3.5×. This rule is exemplified earlier in Table 1. The query consists of a `WHERE` predicate that checks if the rows counts using two different conditions are the same. This is a typical condition especially in auditing or data consistency checking scenarios. When there is a containment relationship between the two conditions, the predicate can be rewritten into a `NOT EXISTS` subquery, which is more efficient as repeated counting is avoided. Similar to the anti-join rewrite above, this rewrite is not covered by the existing rewriter in GaussDB. Thanks to the fine-grained constraint expression in G-DSL, GRewriter can discover this rule and optimize queries of this pattern.

We investigated the presence of these rules in other popular relational databases, including PostgreSQL 16.8, MySQL 8.0, and SQL Server 18.4. The results are shown to the right of Table 5. We use the same 18 queries and, due to confidentiality reason, use synthetic datasets where 10k-row tables are generated from a uniform distribution on their respective schemas. We manually checked the physical plans produced by these databases for the original queries to see if the same rewrites are performed. We validated the result by comparing the execution time of the original and GRewriter-rewritten queries in these databases. PostgreSQL was missing all 7 rules, MySQL was missing 6 rules, and SQL Server was missing 5 rules. This indicates that our rewriting rules are also valuable for other databases. They could integrate GRewriter as a bolt-on rewriter to accelerate these queries.

Runtime Overhead. While GRewriter works as a bolt-on layer to the existing optimizer in GaussDB, it introduces a negligible overhead to the query execution. According to rewriter logs, there are 103 queries with matching rewrite rules. Among them, 73/17/7/6 queries need to explore 1/2/3/4+ rewrite paths, thus adding 1/2/3/4+ calls to the existing optimizer. Due to the manual limit of rewrite iterations, at most 6 paths are explored in a single query execution. As a result, we observed that GRewriter adds 4.5 us on average to the latency of queries that do not match any rules, and 221.7 us to that of queries with matched rules but the query planner considers the rewrite non-beneficial. For other queries, i.e., those that benefit from the rewrite, their latency is reduced as shown in Table 4.

Table 6: Efficiency of different rule generator.

Generator	# of Rule Candidates	Time to Terminate	# of Rules Verified
WeTune	9.4×10^{22}	7 days	1023
WeTune+	2.1×10^{28}	4284 years (est.)	N/A
GRewriter w/o pruning	5.7×10^{29}	116,640 years (est.)	N/A
GRewriter w/ pruning	9.7×10^{22}	8 days	13,973

8.3 Quantity and Quality of Rules

Rule Quantity. The effectiveness of GRewriter heavily depends on how many more complex rules the rule generator can discover compared to existing systems. We evaluate this by comparing with the prior research prototype, WeTune. We also use a configurable denoted as WeTune+, where we increase the number of operators from 6 to the same number as GRewriter, 12. Table 6 shows the numbers of rules discovered and verified, and the time taken. Neither WeTune+ nor GRewriter without pruning could terminate within an acceptable time frame, so the number of candidate rules and termination times provided are estimates. These results are consistent with our expectation that WeTune cannot scale to more

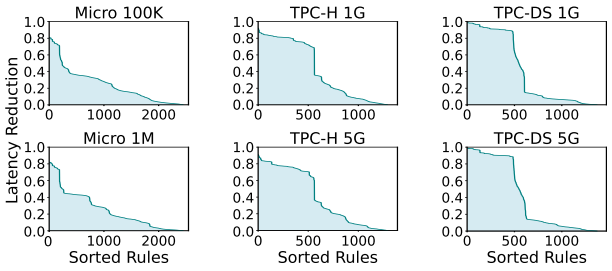

Figure 4: Latency reduction under different datasets.

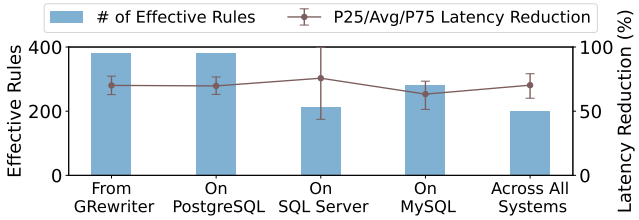

Figure 5: Rules effective on other DBs. Larger numbers indicate greater portions of our rules might be absent in other DBs.

complex rules, and introducing more fine-grained rule representation, G-DSL, would further increase the search space. However, with the set of pruning techniques introduced in Section 6.1, GRewriter can effectively reduce the search space and terminate within 8 days, producing near 14k verified rules. These rules will later be evaluated for their quality and applicability before being integrated into GRewriter's online rewriter. Also thanks to the pruning, the discovery efficiency as measured by the number of verified rules per unit time is increased by more than 10 times.

Rule Quality. We use the six in-house datasets as described in Section 8.1 to evaluate the quality of the rules. Queries that match these rules are generated the same way as described in Section 6.3. The results are shown in Figure 4. For each dataset, we sort the rewrites by latency reduction and plot the ones that bring positive reduction. To select useful rules, we use a threshold of 10% latency reduction. There are 2034 out of the verified 13,973 rules are useful under at least one benchmark and 381 rules are useful in all datasets. The latter portion of rules are more likely to have more applicability, and we use them as the set of rules to be integrated into GaussDB. For each dataset, among the useful rules, the maximum latency reductions are 80.8%, 92.1%, 98.7%, 82.0%, 91.1%, and 98.8%, following the left-to-right, top-to-bottom order shown in Figure 4. More than half of these rules bring a reduction at least 31.8%, 75.2%, 90.2%, 29.4%, 71.1%, and 89.8%, respectively; and the top 10% of the useful rules bring a reduction at least 71.7%, 83.9%, 97.1%, 73.5%, 82.9%, and 97.1%, respectively.

Rule Novelty. We examine the coverage of 381 useful rules in PostgreSQL 16.8, MySQL 8.0, and SQL Server 18.4 using the 1GB TPC-H dataset and the synthesized rewritable queries. We compare the latencies of queries before and after GRewriter's rewrite and report the number of rules that bring a latency reduction exceeding 10% in each system. As shown in Figure 5, 201 rules are useful

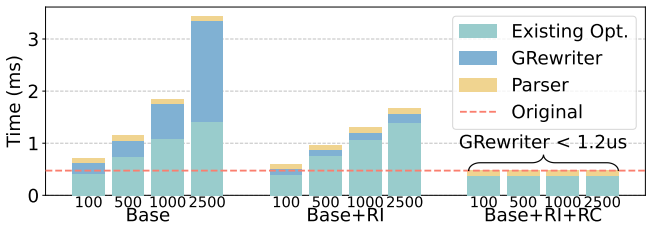

Figure 6: Query optimization time break down under 100, 500, 1000, and 2500 rules. The time spent in the original optimizer includes invocations from GRewriter's rewriter for estimated cost of the rewritten plans.

across all systems, bringing at least a 10% latency reduction with an average reduction of 70.3%. These results indicate that most rules may not be present in these systems, and the bolt-on approach of GRewriter can be used to accelerate these systems as well.

8.4 System Performance Analysis

We evaluated and analyzed the overhead of GRewriter's online rewriter and the impact of Rule Index and Rewrite History Cache. We use the following configurations: (1) `Base`, where GRewriter disables both Rule Index and Rewrite History Cache; (2) `Base+RI`, where only Rule Index is enabled; and (3) `Base+RI+RC`, where both Rule Index and Rewrite History Cache are enabled. As a reference point, we also include the original GaussDB optimizer without GRewriter, denoted as `Original`. Since the overhead of GRewriter is closely related to the number of rules, we load different number of rules sampled from the 13,973 verified rules: 100, 500, 1000, and 2500. We use the standard queries from the TPC-H benchmarks as the workload since these queries are complex, potentially matching many rewrite rules and taking longer to perform the logical plan rewrite. Figure 6 shows the latency breakdown of the whole query optimization process. We break down the time spent on the parser (`Parser`), GRewriter's online rewriter (`GRewriter`), and the existing optimizer (`Existring Opt.`).

Rule Index. Enabling the Rule Index significantly reduces the time of our online rewriter. Without the Rule Index and Rewrite History Cache, the overhead of GRewriter is significant, especially when the number of rules is large. When the number of rules reaches 2500, the `GRewriter` time is reduced by 91.4% compared to when the Rule Index is disabled, and is no larger than 0.17 ms. In terms of overall optimization time, Rule Index has significantly reduced it from 3.32 ms to 1.69 ms with 2500 rules. However, it remains considerably higher than the original GaussDB, mostly spent on the additional invocations to the existing optimizer for physical planning and cost estimation.

Rewrite History Cache. Rewrite History Cache further reduces the query optimization time by caching previous rewrites and their estimated costs. As Figure 6 shows, Rewrite History Cache reduces the time spent on the existing optimizer by 74.3% when the number of rules is 2500. Furthermore, since rewrite paths are now cached, the time previously spent on looking up the Rule Index and applying rewrite is eliminated and replaced by a simple cache lookup. This further reduces time in GRewriter's rewriter to less than 1.2us. With these effects combined, the overall optimization

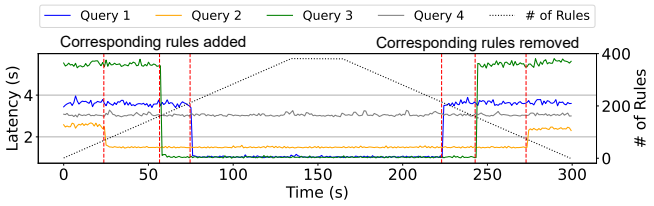

Figure 7: Rule set management's impact on query latency.

time is reduced to 463.2 us, almost identical to the time used by the original optimizer without GRewriter, 462.0 us (0.26% overhead).

Runtime Rule Set Management. We evaluate the impact on query latency of concurrently inserting and removing rules at runtime. We picked three queries that can be optimized by the discovered rules (queries 1, 2, and 3) and one query that cannot be optimized (query 4), and used the 5GB TPC-H dataset. We repeatedly submitted these queries to the server while using another process to insert and remove those 381 rules. As shown in Figure 7, dynamic addition of rules brings immediate improvement to optimizable queries. Regarding impact on irrelevant queries, the average latency of query 4 increases by 0.3% when rule set changed.

9 RELATED WORK

Two-Phase Query Optimization. System R [25] has pioneered the two-phase query optimization approach, which separates logical query rewrite from physical query planning. This idea has influenced many subsequent systems, such as PostgreSQL [16]. However, this approach is inherently static, as rules must be predefined in the code, limiting scalability.

Cascades-Style Query Optimization. To enhance extensibility, cascades-style optimizers can be easily extended with new rules [3, 5, 13–15, 24, 26, 27]. Although they decouple rules, they require encoding new rules with imperative languages, such as Java or Go, and integrating the code into the optimizer. In contrast, rules in G-DSL can be directly loaded into GRewriter without modifying the running database and its optimizer. Additionally, the automated rule discovery in GRewriter complements existing optimizers as discovered rules can further enhance their optimization capabilities.

ML-Based Query Optimization. Recent works [2, 19–21, 28, 34] have made significant progress in ML-based query optimization. Bao [21] and AutoSteer [2] focus on how to select optimization hint sets, which indicate the rewrite rules to be considered during optimization. Given a collection of hint sets, Bao predicts which hint set will be most beneficial using reinforcement learning. AutoSteer further extends Bao with automated hint set discovery, which achieves better optimization capability and can be easily adapted to various databases. LearnedRewrite [34] leverages Monte Carlo Tree Search to explore possible sequences of applying rewrite rules and finds the one with high benefits. LLM-R2 [19] leverages LLMs to recommend rewrite rules. Since these works depend on predefined rewrite rules, GRewriter complements them by automatically discovering new rules, which can further improve their optimization capabilities. Others generate optimized queries or plans without using predefined rewrite rules, such as GenRewrite [20] and BayesQO [28]. Their optimization process can be expensive: GenRewrite needs multiple rounds to rewrite a query, each taking 29.6 seconds on average; BayesQO can take several hours. Thus, they target frequently executed queries. In contrast, GRewriter discovers rules offline and incurs around one millisecond (disabling cache) to rewrite a query at runtime, making it suitable for a wider range of queries.

Program Synthesis for Query Optimization. Program synthesis have been applied to query optimization. QueryBooster [4] automatically generalize given rewrite examples to synthesize rewrite rules. However, it requires human efforts to collect useful examples. To avoid such human efforts, WeTune [32] automatically synthesizes rewrite rules based on rule enumeration and verification. However, it is a research prototype that is not yet ready for use in commercial databases. While GRewriter is inspired by WeTune, it proposes a new DSL and enumeration algorithms to discover more useful rules while maintaining efficiency. Particularly, all useful rules reported in the evaluation cannot be supported by WeTune. GRewriter additionally proposes an efficient online rewriter with a new rule index and a rewrite history cache that has been integrated into GaussDB. SlabCity [11] proposes to directly synthesize optimized queries rather than synthesizing rewrite rules. Due to the large search space of optimized queries, it typically takes nearly a second to rewrite a query. Thus, it targets scenarios where queries are frequently executed (e.g., BI dashboards) or very expensive (e.g., OLAP). Meanwhile, the rule-based rewrite in GRewriter takes around only one millisecond and is applicable in more scenarios.

Middleware Query Optimizer. Recent optimizers have taken a middleware architecture for ease of integration with existing databases. One approach is to interact with user-facing database APIs to execute SQL and explain statements, such as AutoSteer-G [2], GenRewrite [20], and QueryBooster [4]. This approach eliminates the need to modify existing optimizers within the databases. Another approach is to directly modify existing optimizers, which requires more programming efforts but may offer better efficiency. For example, AutoSteer-C [2] has implemented a connector integrated into the optimizer. GRewriter also adopts the latter approach. The main difference is that AutoSteer depends on existing rules, while GRewriter uses newly discovered or crafted rules. GRewriter may complement AutoSteer by connecting to it and enabling a larger optimization space.

Query Equivalence Verification. There has been a long line of work on query equivalence verification [7–10, 31, 33]. They target concrete queries rather than rewrite rules in G-DSL. Thus, we design a rule verifier based on query verifiers.

10 CONCLUSION

GRewriter is a new query rewriter that incorporates hundreds of new rewrite rules and selects suitable ones for given queries efficiently. Rules are automatically generated by a novel generator. GRewriter has offered notable benefits for key GaussDB customers.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (Grant Nos. 62422209, 62132014, 62525202, and 62232009), the National Key R&D Program of China (Grant No. 2023YFB4503600), and Huawei. Zhaoguo Wang is the corresponding author.

REFERENCES

[1] 2025. GRewriter: Practical Query Rewriting with Automatic Rule Set Expansion in GaussDB (Appendix). https://ipads.se.sjtu.edu.cn:1313/seafhttp/f/69dc90c86248426d9376/?op=view.

[2] Christoph Anneser, Nesime Tatbul, David Cohen, Zhenggang Xu, Prithviraj Pandian, Nikolay Laptev, and Ryan Marcus. 2023. AutoSteer: Learned Query Optimization for Any SQL Database. *Proc. VLDB Endow.* 16, 12 (Aug. 2023), 3515–3527. https://doi.org/10.14778/3611540.3611544

[3] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1383–1394. https://doi.org/10.1145/2723372.2742797

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

[5] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J Mior, and Daniel Lemire. 2018. Apache calcite: A foundational framework for optimized query processing over heterogeneous data sources. In *Proceedings of the 2018 International Conference on Management of Data*. 221–230.

[6] Nico Bruno and Cesar Galindo-Legaria. 2021. The Cascades Framework for Query Optimization at Microsoft. https://youtu.be/pQe1LQJiXN0.

[7] Shumo Chu, Daniel Li, Chenglong Wang, Alvin Cheung, and Dan Suciu. 2017. Demonstration of the cosette automated sql prover. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1591–1594.

[8] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (jul 2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[9] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL.. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research* (Chaminade, California, USA) *(CIDR '17)*.

[10] 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. *Proc. ACM Manag. Data* 1, 4, Article 227 (Dec. 2023), 26 pages. https://doi.org/10.1145/3626768

[11] Rui Dong, Jie Liu, Yuxuan Zhu, Cong Yan, Barzan Mozafari, and Xinyu Wang. 2023. SlabCity: Whole-Query Optimization Using Program Synthesis. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3151–3164.

[12] Gartner. 2023. Market Share: All Software Markets, Worldwide, 2022.

[13] Goetz Graefe. 1995. The cascades framework for query optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29.

[14] Goetz Graefe and David J DeWitt. 1987. The EXODUS optimizer generator. In *Proceedings of the 1987 ACM SIGMOD international conference on Management of data*. 160–172.

[15] Goetz Graefe and William J McKenna. 1993. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of IEEE 9th international conference on data engineering*. IEEE, 209–218.

[16] The PostgreSQL Global Development Group. 2024. PostgreSQL. https://www.postgresql.org.

[17] IDC. 2024. 2024 First-Half China Relational Database Software Market Tracking Report (in Chinese).

[18] Guoliang Li, Wengang Tian, Jinyu Zhang, Ronen Grosman, Zongchao Liu, and Sihao Li. 2024. GaussDB: A Cloud-Native Multi-Primary Database with Compute-Memory-Storage Disaggregation. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3786–3798.

[19] 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. *arXiv preprint arXiv:2404.12872* (2024).

[20] Jie Liu and Barzan Mozafari. 2024. Query Rewriting via Large Language Models. arXiv:2403.09060 [cs.DB] https://arxiv.org/abs/2403.09060

[21] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 1275–1288. https://doi.org/10.1145/3448016.3452838

[22] Microsoft. 2024. SQL Server. https://www.microsoft.com/sql-server.

[23] Oracle. 2024. MySQL. https://www.mysql.com.

[24] Hamid Pirahesh, Joseph M Hellerstein, and Waqar Hasan. 1992. Extensible/rule based query rewrite optimization in Starburst. *ACM Sigmod Record* 21, 2 (1992), 39–48.

[25] P Griffiths Selinger, Morton M Astrahan, Donald D Chamberlin, Raymond A Lorie, and Thomas G Price. 1979. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*. 23–34.

[26] Mohamed A Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, et al. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 337–348.

[27] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. 2020. Cockroachdb: The resilient geo-distributed sql database. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1493–1509.

[28] Jeffrey Tao, Natalie Maus, Haydn Jones, Yimeng Zeng, Jacob R. Gardner, and Ryan Marcus. 2025. Learned Offline Query Planning via Bayesian Optimization. arXiv:2502.05256 [cs.DB] https://arxiv.org/abs/2502.05256

[29] TPC. 2024. TPC-DS. https://www.tpc.org/tpcds/.

[30] TPC. 2024. TPC-H. https://www.tpc.org/tpch/.

[31] Shuxian Wang, Sicheng Pan, and Alvin Cheung. 2024. QED: A Powerful Query Equivalence Decider for SQL. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 3602–3614. https://doi.org/10.14778/3681954.3682024

[32] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic discovery and verification of query rewrite rules. In *Proceedings of the 2022 International Conference on Management of Data*. 94–107.

[33] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. (2022), 2735–2748. https://doi.org/10.1109/ICDE53745.2022.00250

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