QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting

Qiushi Bai
University of California, Irvine
qbai1@uci.edu

Sadeem Alsudais
University of California, Irvine
salsudai@uci.edu

Chen Li
University of California, Irvine
chenli@ics.uci.edu

ABSTRACT

SQL query performance is critical in database applications, and query rewriting is a technique that transforms an original query into an equivalent query with a better performance. In a wide range of database-supported systems, there is a unique problem where both the application and database layer are black boxes, and the developers need to use their knowledge about the data and domain to rewrite queries sent from the application to the database for better performance. Unfortunately, existing solutions do not give the users enough freedom to express their rewriting needs. To address this problem, we propose QueryBooster, a novel middleware-based service architecture for human-centered query rewriting, where users can use its expressive and easy-to-use rule language (called VarSQL) to formulate rewriting rules based on their needs. It also allows users to express rewriting intentions by providing examples of the original query and its rewritten query. QueryBooster automatically generalizes them to rewriting rules and suggests high-quality ones. We conduct a user study to show the benefits of VarSQL to formulate rewriting rules. Our experiments on real and synthetic workloads show the effectiveness of the rule-suggesting framework and the significant advantages of using QueryBooster for human-centered query rewriting to improve the end-to-end query performance.

PVLDB Reference Format:
Qiushi Bai, Sadeem Alsudais, and Chen Li. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. PVLDB, 16(11): 2911 - 2924, 2023.
doi:10.14778/3611479.3611497

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ISG-ICS/QueryBooster.

1 INTRODUCTION

System performance is critical in many database applications where users need answers quickly to gain timely insights and make mission-critical decisions. In the large body of optimization literature [16, 25, 33], one family of technique is query rewriting, which transforms a query to a new query that computes the same answers with a higher performance.

Proceedings of the VLDB Endowment, Vol. 16, No. 11 ISSN 2150-8097.
doi:10.14778/3611479.3611497

Motivating example. Figure 1 shows a case where a user runs Tableau on top of a Postgres database to analyze and visualize the underlying data of social media tweets. Tableau formulates and sends a SQL query to the database for each frontend request through a connector such as a JDBC driver. The database returns the result to Tableau to render in the frontend.

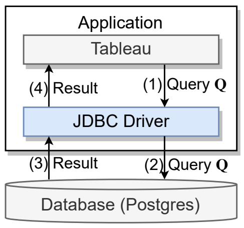


Figure 1: Query lifecycle between Tableau and Postgres.

Figure 2a shows an example SQL query Q formulated by Tableau to compute a choropleth map of tweets containing a substring, e.g., `covid`, which matches `covid-19`, `covid19`, `postcovid`, `covidvaccine`, etc. Without any index available on the table, the database engine uses a scan-based physical plan, which takes 34 seconds in our evaluation. To improve the performance, the developer is tempted to create an index on the `content` attribute of the table.

```
SELECT SUM(1) AS "cnt:tweets",
       "state_name" AS "state_name"
FROM "tweets"
WHERE STRPOS(LOWER("content"),
       'covid') > 0
GROUP BY 2;
```

(a) An original query Q formulated by Tableau to compute a choropleth map of tweets containing *covid* as a substring.

```
SELECT SUM(1) AS "cnt:tweets",
       "state_name" AS "state_name"
FROM "tweets"
WHERE "content" ILIKE
       '%covid%'
GROUP BY 2;
```

(b) A rewritten query Q' equivalent to Q but runs 100 times faster by using a trigram index on the `content` attribute.

Figure 2: An example query pair (differences shown in blue).

Unfortunately, Postgres does not support an index-based physical plan for the `STRPOS(LOWER("content"),`s`)` expression in Q, where s is an arbitrary string. Interestingly, another query Q', shown in Figure 2b, is equivalent to Q, and uses an `ILIKE` predicate. This expression can be answered using a trigram index on the `content` attribute [37], and the corresponding physical plan takes 0.32 seconds only. Notice that the optimizer does not produce an index-based plan for the original `STRPOS` predicate using this trigram index [41].

A natural question is whether we can let Tableau generate Q' instead of Q for the database. Tableau is a proprietary application layer, and has its own internal logic to generate queries, which the developer, in this example, cannot change. We may also consider using the `CREATE RULE` interface provided by Postgres [35] to introduce a rewriting rule inside the database, but as we will show in

Section 2, this language has limited expressive power and does not allow us to rewrite Q to Q'. As a consequence, we miss the rewriting opportunity to significantly improve the query performance. Note that as shown in Section 7.5, the rewriting need is not limited to simple predicate levels but also includes complex statement levels.

Problem Formulation. Besides the above example, as more cases in Section 2 and our experiments using different applications and databases on both synthetic and real-world datasets in Section 7.5 show, there is a unique problem in a wide range of database-supported systems with the following setting. (1) *The developers need to treat the application layer as a black box and cannot modify its logic of generating SQL queries.* Reasons include i) the application is proprietary software (e.g., Tableau); and ii) the source code of the application is too complicated or old to modify, especially for legacy systems [24]. For example, reports [49] show that there are many applications where parties have even lost their original source code. (2) *The developers need to treat the database as a black box.* Reasons include i) the developers do not have the privileges to modify the database; and ii) the database is used by many clients, and the developers want to avoid side effects of database changes to these clients. (3) *The developers want to use their knowledge about the data and domain to rewrite queries sent from the application to the database to significantly improve their performance.* For example, they may introduce rewriting rules that are valid for their particular database with certain properties (e.g., specific attribute types or certain cardinality constraints), even though these rules may not be valid for all databases. Specifically, the experimental results in Section 7.5 illustrate cases where a rewriting is valid only for a particular dataset, and may not be correct in general, thus it cannot be adopted by a database query optimizer. Thus, we want to allow developers to be "in the driver's seat" during the lifecycle of a query to generate an equivalent and more efficient query as "human-centered query rewriting". Note that we do not seek to replace query optimizers inside databases but only provide a chance for users to inject their knowledge to optimize queries before they are sent to the database. Hence, the problem is stated as:

Problem Statement: Given an application and a database as black boxes, develop a middleware solution for users to easily express their rules to rewrite application queries for a better performance.

Solution overview. In this paper, we propose QueryBooster, a novel middleware-based service architecture for human-centered query rewriting. It is between an application and a database, intercepts SQL queries sent from the application, and rewrites them using human-crafted rewriting rules to improve their performance. By providing a slightly-modified JDBC/ODBC driver or a RESTful proxy for the query interception, QueryBooster requires no code changes to either the application or the database. QueryBooster provides an expressive and easy-to-use rule language (called VarSQL) for users (SQL developers or DBAs) to define rewriting rules (i.e., customizing query rewriting for their application queries). Users can easily express their rewriting needs by providing the query pattern and its replacement. They can also specify additional constraints and actions for complex rewriting details. In addition, QueryBooster allows users to express their rewriting intentions by providing examples. That is, users can input original queries and the desired rewritten queries. Then QueryBooster automatically generalizes the examples into rewriting rules and suggests high-quality rules. The users can confirm the rules to be saved in the system or further modify the rules as they want.

Challenges and contributions. To develop the QueryBooster system, we face several challenges. *(C1)* How to develop an expressive and easy-to-use rule language for users to formulate rules? *(C2)* How to generalize pairs of original and rewritten queries to rewriting rules and measure their quality? *(C3)* How to search the candidate rewriting rules to suggest high-quality ones based on the user-given examples? In this paper, we study these challenges and make the following contributions.

- We propose a novel middleware-based query rewriting service to fulfill the need of human-centered query rewriting (Section 3).
- We study the suitability of existing rule languages in the literature and show their limitations. We then develop a novel rule language (VarSQL) that is expressive and easy to use (Section 4).
- We develop transformations to generalize pairs of rewriting queries to rules and propose using the metric of minimum description length to measure rule quality (Section 5).
- We present a framework to search the candidate rewriting rules efficiently and suggest high-quality rules based on user-given examples (Section 6).
- We conduct a thorough experimental evaluation, including a user study, to show the benefits of the VarSQL rule language, the effectiveness of the rule-suggesting framework, and the advantages of human-centered query rewriting (Section 7).

2 RELATED WORK AND LIMITATIONS

In this section, we show existing solutions and why they cannot solve the formulated problem. Figure 3 gives an overview of various solutions for query rewriting in the lifecycle of a query in a database system [2, 6, 12, 27, 54, 56] and the position of the proposed QueryBooster system. At a high level, these solutions can be classified into two categories: native writing plugins and third-party solutions.

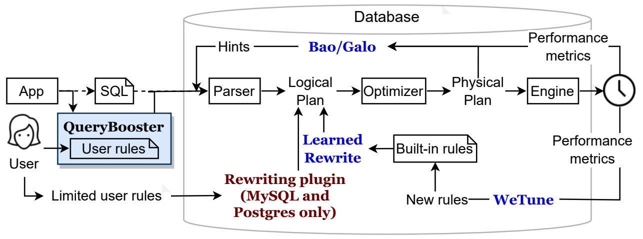


Figure 3: Query-rewriting solutions for databases (native solutions in brown and third-party solutions in blue).

Native rewriting plugins. Most databases such as AsterixDB [1], IBM DB2 [21], MongoDB [28], MS SQL Server [50], MySQL [29], Oracle [31], Postgres [35], SAP HANA [45], Snowflake [48], and Teradata [39], do not have capabilities for users to rewrite queries sent to the database. Notice that even though "hints" can be included in a query to make suggestions to the database optimizer, they are technically not used to change the query, thus, are not a query-rewriting solution. To our best knowledge, only two database systems, Postgres and MySQL, provide a plugin for users to define new rules to rewrite queries before sending them to the database. However, their rule-definition languages have limited expressive power, as discussed below.

Postgres. A rewriting rule in the Postgres plugin can only define a pattern matching a table name in a `SELECT` clause of a SQL query and replace the table with another table or a subquery [35]. Its rule language cannot express the rewriting in the running example in Figure 2. In particular, it does not support a pattern that matches a component in a SQL statement at the predicate level, e.g., the `STRPOS(LOWER("content"), _) > 0` portion in the `WHERE` clause in the original query. Safety could be a major consideration behind this rule language. For instance, the Postgres 14 documentation [36] explained that "*this restriction was required to make rules safe enough to open them for ordinary users, and it restricts ON SELECT rules to act like views.*"

MySQL. The MySQL plugin uses the syntax of prepared statements to define query-rewriting rules, and a rule replaces a SQL query matching the rule's pattern with a new statement [29]. A rule includes placeholders that can only match literal values in a SQL query, such as a constant in a predicate in the `WHERE` clause. A main limitation of this language is that a placeholder cannot match many components in a query, such as table names and attribute names. For instance, the following is a predicate from a query formulated by Tableau to MySQL:

```
adddate(date_format(`created_at`, '%Y-%m-01 00:00:00'),
        interval 0 second) = TIMESTAMP('2018-04-01 00:00:00')
```

And if we rewrite the predicate by removing the type-casting on the right-hand constant, as shown below:

```
adddate(date_format('created_at', '%Y-%m-01 00:00:00'),
        interval 0 second) = '2018-04-01 00:00:00'
```

The corresponding rewritten query is significantly faster (2.68s) than the original query (87s). Unfortunately, the MySQL plugin does not support this rewriting because a pattern in the MySQL plugin has to be an *entire* statement instead of a *single* predicate. In other words, using the MySQL plugin for this rewriting requires the enumeration of all other parts of the target SQL query.

Third-party solutions. Bao [27] and Galo [12] rewrite queries by adding hints to help the database optimizer generate more efficient physical plans based on their cost estimations and searching methods. They take a physical plan and query performance as the input and produce hints to the original query. WeTune [54] generates new rewriting rules automatically by searching the logical-plan space and considering the performance of rewritten queries. LearnedRewrite [56] utilizes built-in rewriting rules inside the database to optimize queries, and the users have no control over when and which rules are applied. None of these solutions allow users to formulate their own rewriting rules to fulfill the human-centered query rewriting need. PgCuckoo [20] opens an opportunity for users to inject intelligent logic to manipulate query plans in Postgres. It only works for Postgres and the proposed middleware solution works for any databases.

Commercial systems. There are also commercial systems that do query rewriting for applications on top of databases. For example, Keebo [23] uses data learning and *approximate* query processing (AQP) techniques to accelerate analytical queries. It runs queries on summarized tables instead of the raw data as much as possible to reduce query time. EverSQL [15] uses AI/ML techniques to recommend rewriting ideas for queries on MySQL and Postgres. Other systems such as ApexSQL [4], Query Performance Insights for Azure SQL [38], and Toad [52] help database developers analyze query performance bottlenecks and tune database knobs. None of these systems allow users to formulate their own rewriting rules to fulfill the human-centered query rewriting need.

General pattern-matching tools. These tools can be used to rewrite any program and are not limited to SQL code. For instance, Quasiquotation [32, 46] is a general technique to rewrite programs using meta-programs. A main issue of the tools is that they are not designed for SQL queries, and they do not consider the unique semantics (tables, columns, etc.) of SQL, which is considered by the proposed QueryBooster.

3 QUERYBOOSTER: OVERVIEW

We present a novel middleware system called QueryBooster, to fulfill the need for human-centered query rewriting.

Figure 4 shows the architecture of QueryBooster. It includes two phases, an offline *rule formulation* phase and an online *query rewriting* phase.

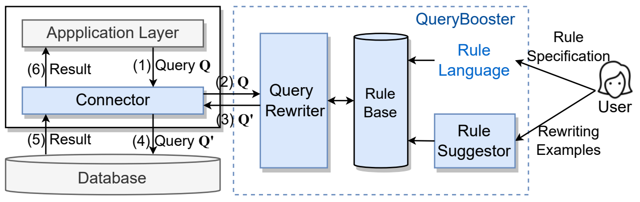


Figure 4: Architecture of QueryBooster.

For the offline *rule formulation* phase, QueryBooster provides a powerful interface for users to formulate rewriting rules. It allows users to formulate rules in the following two ways. First, it provides an expressive and easy-to-use rule language for users to define rewriting rules. Users can easily express their rewriting needs by writing down the query pattern and its replacement. They can also specify additional constraints and actions to express complex rewriting details. Second, it allows users to express their rewriting intentions by providing examples. A rewriting example is a pair of SQL queries with the original query and the desired rewritten query. The "Rule Suggestor" automatically suggests high-quality rewriting rules based on the examples. The users can choose their desired rewriting rules and further modify suggested rules as they want. All user-confirmed rules are stored in the "Rule Base," and the "Query Rewriter" will rewrite online queries based on the rules.

For the online *query rewriting* phase, QueryBooster provides a customized connector that communicates with its service to rewrite application queries. In particular, the connector accepts an original query Q formulated by the application and sends Q to the "Query Rewriter" service, which applies rewriting rules stored in the "Rule Base" to rewrite Q to a new query Q'. The new query is sent back to the connector, which forwards Q' to the backend database to boost the application's performance. Note that QueryBooster focuses on rewriting queries based on user-specified rules and assumes no access to the backend database to create indexes.

To use the QueryBooster rewriting service, users do not need to modify any code of the applications or databases or install any plugins. They only need to replace the existing DB connector with a QueryBooster-customized one. The connector can be for either an ODBC/JDBC interface or a RESTful interface. Most database vendors provide ODBC/JDBC drivers with an open-source license. Thus we can provide a slightly modified version of the driver that communicates with the proposed QueryBooster service to rewrite queries for these databases. For instance, in our developed prototype [5], we added only 112 lines of code to the PostgreSQL JDBC driver. For databases with redistribution restrictions on their drivers (e.g., Oracle JDBC driver [30]), we can provide users a software patch with a small amount of source code modifications. For applications and databases that communicate through a RESTful interface, we can provide a proxy web server that forwards all requests and responses between them transparently. The proxy server in the middle rewrites an application request by communicating with the Query Rewriter service. We assume the RESTful API endpoint in the application is configurable, i.e., we can switch the target database endpoint to our service.

Correctness of rewriting rules. In the case where users make mistakes when formulating rewriting rules, we can leverage existing query equivalence verifiers (e.g., [10, 54]) to validate the rules and guarantee their correctness.

4 VARSQL: A REWRITING-RULE LANGUAGE

The main task of QueryBooster is to provide an expressive and easy-to-use rule language that meet the following three requirements. *(R1) Powerful expressiveness in SQL semantics.* It needs to understand SQL-specific semantics where users can specify pattern-matching conditions on the elements of a SQL query, e.g., two tables have the same name. *(R2) Easy to use by SQL users.* Users of QueryBooster are application developers who are familiar with SQL. QueryBooster should require users to have little prior knowledge other than SQL to define their rewriting rules. *(R3) Independent of databases or SQL dialects.* As a general query-rewriting service, the rule language should be independent of any specific database or SQL dialect.

In this section, we first study the suitability of existing rule languages in the literature and then develop a novel rule language that meets all the requirements desired by QueryBooster.

4.1 Suitability of Existing Rule Languages

Existing rule languages [2, 9, 11, 13, 16, 18, 33, 34, 47, 55] are summarized in Figure 5.

General versus SQL-specific. The languages on the left are more general (i.e., non-SQL-specific) since they have fewer SQL-specific restrictions. For example, Regex [55] does not require a SQL query as the input, while EDS [16] only accepts valid SQL query plans. The main advantage of SQL-specific languages is that they are very powerful for users to express rewriting rules in SQL-specific semantics. For instance, we do not need to specify SQL-specific syntax requirements such as white spaces, and we can specify SQL-specific constraints for variables that are not supported by general languages, such as "x is a column."

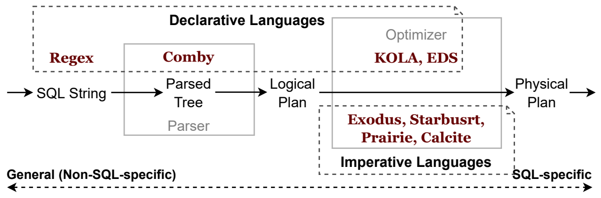


Figure 5: Existing rule languages (shown in brown) in the lifecycle of a SQL query.

There are also disadvantages of the more specific languages. First, they can be limited to a particular SQL dialect or database. For instance, EDS is designed for a particular extensible system (called "EDBMS") and its SQL dialect [16]. Second, they require the users to deeply understand how a SQL query is translated into a plan and how the database optimizer works.

Declarative versus Imperative. The existing rule languages are either *declarative* (e.g., Regex) or *imperative* (e.g., Calcite). The primary disadvantage of using an imperative language to define rules is that it requires users to have prior knowledge about the internal structures of the rule engine and define rules by writing code. For example, in Calcite, a user has to write a Java class that implements an interface to define a new rule.

The main advantage of imperative languages is the expressive power offered by the programming language (e.g., C++), such as defining schema-dependent pattern-matching conditions. For example, a rewriting rule that removes unnecessary self-joins may need to verify the joining attribute is unique, which cannot be inferred just from the SQL query itself. Using an imperative rule engine, we can easily write a rule with a few lines of C code [34] that accesses the schema data and checks the matching condition.

Table 1: Suitability of existing languages for QueryBooster.

Rule Language	Expressive Power		Independent of DB	Additional Knowledge Users Need
	SQL Semantics	*SQL Schema*		
Regex	No	No	**Yes**	
Comby	No	No	**Yes**	
KOLA	**Yes**	No	**Yes**	①②③
EDS	**Yes**	**Yes**	No	①②
Exodus	**Yes**	**Yes**	No	①②
Starburst	**Yes**	**Yes**	No	①②④⑤
Prairie	**Yes**	**Yes**	No	①②⑤
Calcite	**Yes**	**Yes**	**Yes**	①②④⑥
VarSQL	**Yes**	**Yes**	**Yes**	

① Query Optimization; ② Relational Algebra; ③ Combinator-based Algebra; ④ Internal Data Structure; ⑤ C++ Programming; ⑥ Java Programming;

To this end, we summarize how existing languages meet QueryBooster's requirements on its rule language in Table 1. An observation is that no existing rule language satisfies all the requirements. Next, we develop a novel rewriting-rule language called VarSQL.

4.2 VarSQL: A Novel Rule Language

We develop a novel rewriting-rule language (called VarSQL[1]) for QueryBooster that meets all the requirements. In particular, VarSQL understands SQL-specific semantics and supports schema-dependent pattern-matching conditions (R1). It is easy to use, requiring no prior knowledge other than SQL (R2). Also, it is independent of any specific database or SQL dialect (R3). Next, we present the technical details of VarSQL.

The syntax of VarSQL to define a rewriting rule is as follows:

[Rule] ::= **[Pattern]** / **[Constraints]** --> **[Replacement]** / **[Actions]**.

VarSQL uses a four-component structure adopted by most rule languages (e.g., EDS [16] and Comby [11]). The "Pattern" and "Replacement" components define how a query is matched and rewritten into a new query. The "Constraints" component defines additional conditions that cannot be specified by a pattern such as schema-dependent conditions. The "Actions" component defines extra operations that the replacement cannot express, such as replacing a table's references with another table's. We first discuss how a pattern and a replacement are formulated using VarSQL.

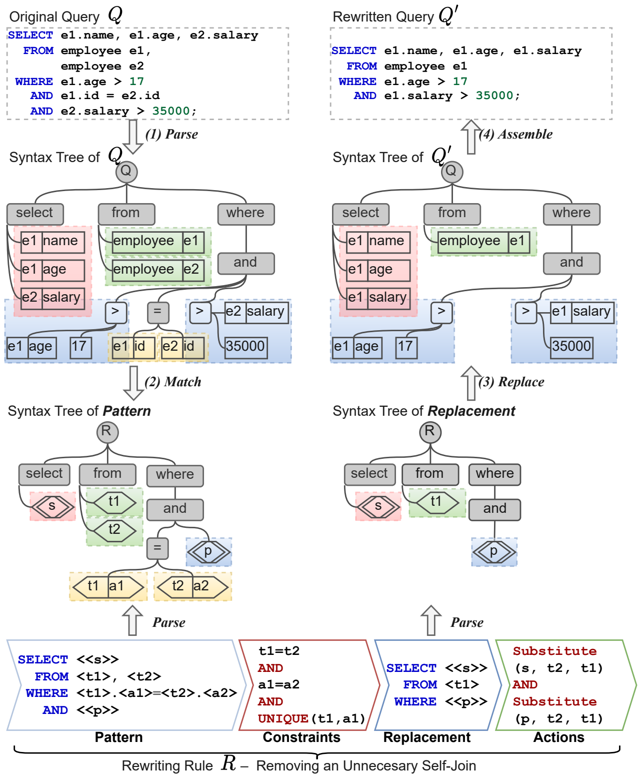
Extending SQL with variables. The main idea of using VarSQL to define a rule's pattern is to extend the SQL language with variables. A variable in a SQL query pattern can represent an existing SQL element such as a table, a column, a value, an expression, a predicate, a sub-query, etc. In this way, a user can formulate a query pattern as easily as writing a normal SQL query. The only difference is that, using VarSQL, one can use a variable to represent a specific SQL element so that the pattern can match a broad set of SQL queries. We call this pattern-formulating process "variablizing" a SQL query, and we call the formulated pattern query a "variablized" SQL query. Similarly, a rule's replacement is formulated by writing the rewritten SQL query using variables introduced in the rule's pattern. Particularly, both the pattern and replacement in a VarSQL rule have to be a full or partial SQL query optionally variablized. The variables and their matching conditions are defined in Table 2.

Table 2: Variable definitions in VarSQL.

Name	Syntax (regex)	Description	Example
Element-Variable	<[a-zA-Z0-9_]*>	An element-variable matches a table, a column, a value, an expression, a predicate, or a sub-query.	`STRPOS(LOWER(<x>), 'iphone') > 0` <x> matches any value, column, expression, or sub-query.
Set-Variable	<<[a-zA-Z0-9_]*>>	A set-variable matches a set of tables, columns, values, expressions, predicates, or sub-queries.	`SELECT <<x>> FROM <t> WHERE <<p>>` <<x>> matches any set of values, columns, expressions, or sub-queries.

SQL syntax tree-based pattern matching and replacement. VarSQL does the pattern matching and replacement at the SQL

[1]VarSQL stands for "Variablized SQL".



```
SELECT e1.name, e1.age, e2.salary
  FROM employee e1,
       employee e2
 WHERE e1.age > 17
   AND e1.id = e2.id
   AND e2.salary > 35000;
```

```
SELECT e1.name, e1.age, e1.salary
  FROM employee e1
 WHERE e1.age > 17
   AND e1.salary > 35000;
```

```
SELECT <<s>>
  FROM <t1>, <t2>
 WHERE <t1>.<a1>=<t2>.<a2>
   AND <<p>>
```

```
t1=t2
AND
a1=a2
AND
UNIQUE(t1,a1)
```

```
SELECT <<s>>
  FROM <t1>
 WHERE <<p>>
```

```
Substitute
(s, t2, t1)
AND
Substitute
(p, t2, t1)
```

Figure 6: The process of pattern matching and replacing of a VarSQL rule R on an example query Q. The gray nodes in both syntax trees of Q and the R's pattern are matched keywords. The colored dashed boxes show the variables in R's pattern and their matched elements in Q.

syntax tree level. Consider the rule R shown at the bottom of Figure 6, where the pattern and replacement are also shown in their syntax tree formats. This rule specifies that when two tables with the same name join on the same unique column, we can safely remove the join and keep only one copy of the table. The remaining of Figure 6 shows the process of pattern matching and replacement of this rule on an example query Q. We first obtain the syntax tree of the query, and compare it node by node against the syntax tree of the rule's pattern. The keyword nodes match each other, and the variables in the pattern match those elements in the query. Under the node "and", the subtree "`<t1>.<a1>=<t2>.<a2>`" in the pattern matches the predicate "`e1.id=e2.id`" in the query, and the set-variable "<<p>>" matches the two remaining predicates in the query. Next, we use the rule's replacement syntax tree as a template to generate the rewritten query's syntax tree by replacing the variables with their matched elements in the pattern. Finally, we assemble the rewritten query from the syntax tree.

Providing pre-implemented imperative procedures. Based on SQL, VarSQL is a declarative language. One problem with declarative languages is that they lack the expressive power to define complex logic in the replacement of a rule and schema-dependent pattern-matching conditions where imperative programs are needed to access the database schema. To solve this issue, VarSQL adopts the idea used in declarative languages such as EDS and Comby that it provides pre-implemented imperative procedures for users to define complex logic in the constraint and action components of rules. For example, the last constraint "`UNIQUE(t1, a1)`" defined in the "Constraints" component in the rule shown in Figure 6 calls the pre-implemented imperative procedure "`UNIQUE`" supported by VarSQL, which verifies if "`a1`" in table "`t1`" is a unique column by referring to the database schema.

VarSQL also provides imperative procedures for users to define complex actions in a rule. For example, the rule in Figure 6 does two actions on the replacement SQL query. The first action "`Substitute(s, t2, t1)`" is to replace the table "`t2`" with table "`t1`" in the scope represented by the set-variable "<<s>>". Consider the query Q in Figure 6 that matches the rule. The set-variable "<<s>>" matches the entire selection list "`e1.name, e1.age, e2.salary`". Since the replacement of the rule removes table "`t2`" from the query, the column "`e2.salary`" needs to be substituted by "`e1.salary`". And, the action "`Substitute(s, t2, t1)`" achieves this purpose.

To make sure the pattern-matching and replacement at the syntax tree level can handle SQL semantics, VarSQL understands important SQL concepts, e.g., an element-variable "<x>" in the `FROM` clause can match either a table name or a table name with an alias.

5 RULE QUALITY AND TRANSFORMATIONS

In this section, we focus on providing a powerful interface for QueryBooster that suggests high-quality rules for user-given rewriting examples. We first discuss how to measure the quality of rules and formally define the rewriting-rule suggestion problem. We then propose a framework to solve the problem, which comprises two major steps: transforming rules into more general forms and searching for high-quality rules greedily. We discuss the first step in this section and the second step in the next section.

5.1 Quality of Rewriting Rules

When the rule suggestor generates rules from the user-given examples, there can be many different sets of rules that can achieve the example rewritings. For instance, consider the five input examples in Figure 7. The rule suggestor can output the original five rewriting pairs as five rules to the user. Apparently, this suggestion is an overfit to the given examples since the suggested rules cannot rewrite queries slightly different from the examples. Intuitively, we want to suggest more general rules that capture the pattern of the given examples. At the same time, we do not want to over-generalize the rules, which may underfit the examples. For instance, in Figure 7, both rules r_2 and r_3 can achieve the rewritings for the example pairs (Q_4, Q_4') and (Q_5, Q_5'), which removes the `ORDER BY` clause from the subquery. In this case, r_2 is less general than r_3 but is a better suggestion, because lacking the context of a `COUNT` aggregation in the outer query, r_3 can be erroneous in many cases.

To this end, we want to avoid underfitting or overfitting the given examples when measuring the quality of rewriting rules. An effective way is through the Minimum Description Length (MDL) principle [43], which minimizes the total length required to describe the underlying patterns in the data. There are MDL-based metrics for pattern extractions in domains such as data mining [17], data cleaning [19, 40], and regex learning [7]. We can adapt these existing metrics to measure our rewriting rules' quality or derive our own description length functions as needed. From the rule-suggestor's perspective, we assume a rule-quality metric is given.

For the MDL metric, we assume no access to the target database. If we are granted access, we can also consider the rewriting rules' effectiveness in improving the performance of the historical workload as the rules' quality. For simplicity, we first use MDL as the quality function and then discuss how to extend the framework to include query performance to measure the rules' quality in Section 6.3.

Rewriting-rule suggestion problem. Next, we formally define the problem of suggesting high-quality rules from given examples.

Definition 5.1. (Covering) Let Q be a set of query rewriting pairs $\{(Q_1, Q_1'), (Q_2, Q_2'), \ldots, (Q_n, Q_n')\}$, and R be a set of rewriting rules $\{r_1, r_2, \ldots, r_k\}$. We say R *covers* Q if for each pair (Q_i, Q_i') in Q, there is at least one rule r_j in R such that r_j can rewrite Q_i into Q_i', and there is no rule r_k in R such that r_k can rewrite Q_i into a query different than Q_i'.

Definition 5.2. (Rewriting-rule suggestion problem) Let Q be a given set of query rewriting pairs $\{(Q_1, Q_1'), (Q_2, Q_2'), \ldots, (Q_n, Q_n')\}$, G be a given rule language, and L be a given description length function. The *rewriting-rule suggestion problem* is to compute a set R of rewriting rules $\{r_1, r_2, \ldots, r_k\}$ written in G such that R covers Q and the total length of rules $\Sigma_{i=1\ldots k} L(r_i)$ is minimal.

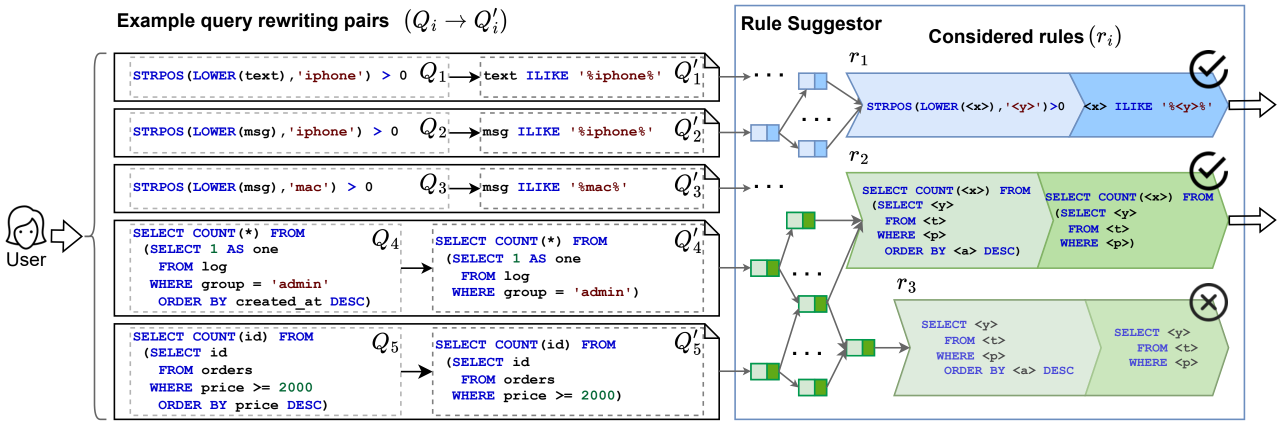
We propose a two-step solution. First, we define a set of transformations that can generalize a rewriting rule into a more general form such that the transformed rule can cover more rewriting pairs than the original rule. By applying the transformations on the given rewriting pairs iteratively, we identify a set of candidate rules to consider for the final suggestion. Second, we adopt a greedy-search strategy to efficiently explore different subsets of rules as candidates and minimize the total description length. Next, we present the technical details of both steps.

5.2 Transforming Rules to More General Forms

Transformations on rules. A transformation on a rewriting rule can generalize the rule into a more general rule such that the new rule covers more rewriting pairs than the original one. The instantiation of transformations is dependent on the given rule language. We now define transformations (shown in Figure 8) on rewriting rules formulated in the VarSQL language, namely *Variablize-a-Leaf*, *Variablize-a-Subtree*, *Merge-Variables*, and *Drop-a-Branch*. The last three transformations only happen if the replaced variables are not referred to in other places in the rule's pattern or replacement.

Variablize-a-Leaf. This transformation replaces an instantiated element (table, column, or value) in a rule with a variable. In this way, the transformed rule can match more queries than the original one. As shown in the first example in Figure 8, the transformed rule can match a query with any column name in the first argument of the `STRPOS` function. In contrast, the original rule can only match a query with the specific "msg" column.

Variablize-a-Subtree. This transformation replaces a complex element (expression, predicate, or subquery) in a rule with a variable. In this way, we can generalize the pattern of a rule by hiding the details within an expression, predicate, or subquery. In the second



```
Q1:  STRPOS(LOWER(text),'iphone') > 0        Q'1: text ILIKE '%iphone%'
Q2:  STRPOS(LOWER(msg),'iphone') > 0         Q'2: msg ILIKE '%iphone%'
Q3:  STRPOS(LOWER(msg),'mac') > 0            Q'3: msg ILIKE '%mac%'
Q4:  SELECT COUNT(*) FROM                    Q'4: SELECT COUNT(*) FROM
       (SELECT 1 AS one                             (SELECT 1 AS one
        FROM log                                     FROM log
        WHERE group = 'admin'                        WHERE group = 'admin')
        ORDER BY created_at DESC)
Q5:  SELECT COUNT(id) FROM                   Q'5: SELECT COUNT(id) FROM
       (SELECT id                                   (SELECT id
        FROM orders                                  FROM orders
        WHERE price >= 2000                          WHERE price >= 2000)
        ORDER BY price DESC)

r1: STRPOS(LOWER(<x>),'<y>')>0   ->   <x> ILIKE '%<y>%'
r2: SELECT COUNT(<x>) FROM                    ->  SELECT COUNT(<x>) FROM
      (SELECT <y>                                   (SELECT <y>
       FROM <t>                                      FROM <t>
       WHERE <p>                                     WHERE <p>)
       ORDER BY <a> DESC)
r3: SELECT <y>                                ->  SELECT <y>
    FROM <t>                                      FROM <t>
    WHERE <p>                                     WHERE <p>
    ORDER BY <a> DESC
```

Figure 7: Suggesting rewriting rules from user-given examples. The rule suggestor suggests two rewriting rules (r_1 and r_2) that cover all five query rewriting pairs provided by the user, and the total description length of r_1 and r_2 is minimized compared to other suggestions.

Transformation	Description	Example
Variablize-a-Leaf	Notate a leaf in a rule's pattern AST as a variable. A leaf has to be a table, column or value.	STRPOS(msg,'mac')>0 msg LIKE '%mac%' ↓ STRPOS(<x>),'mac')>0 <x> ILIK '%mac%'
Variablize-a-Subtree	Notate a common subtree in both a rule's pattern and replacement ASTs as a variable. A subtree has to satisfy: (1) its height is one; (2) none of children are a table, column or value; (3) the root is not a clause-leading keyword such as *select, from, where*, etc.	CAST(DATE_TRUNC('day', CAST(<x> AS DATE)) AS DATE) DATE_TRUNC('day', CAST(<x> AS DATE)) ↓ CAST(DATE_TRUNC('day', <y>) AS DATE) DATE_TRUNC('day', <y>)
Merge-Variables	Merge a common set of sibling variables in both a rule's pattern and replacement ASTs into one variable.	SELECT <a>, <b> FROM <t1>, <t2> WHERE <t1>.<c> = <t2>.<c> SELECT <a>, <b> FROM <t1> ↓ SELECT <<s>> FROM <t1>, <t2> WHERE <t1>.<c> = <t2>.<c> SELECT <<s>> FROM <t1>
Drop-a-Branch	Remove a common branch (starting from the root) in both a rule's pattern and replacement ASTs.	SELECT <a> FROM <t> WHERE STRPOS(<x>),'<y>')>0 SELECT <a> FROM <t> WHERE <x> LIKE '%<y>%' ↓ FROM <t> WHERE STRPOS(<x>),'<y>')>0 FROM <t> WHERE <x> LIKE '%<y>%'

Figure 8: Transformations on rewriting rules formulated in VarSQL. A transformation is applied to the pattern and replacement ASTs of a rewriting rule to generalize it into a more general rule.

example in Figure 8, the common expression "`CAST(<x> AS DATE)`" appears without any modifications in both the rule's pattern and replacement, which means that it might be an irrelevant pattern in the original rule. Summarizing the common expression with a new variable makes the rule more general.

Merge-Variables. Notice that in the VarSQL language, an element-variable can only match a single element in queries. We introduce this transformation to generalize a set of variables to a set-variable to suppress the quantity restriction when matching queries. As shown in the third example in Figure 8, the original rule only matches queries with the two columns in the `SELECT` clause, and the transformed rule can match queries with any number of columns in the selection list. This transformation is useful when we want a more general rule where the quantity of elements does not matter for the pattern.

Drop-a-Branch. This transformation is a complement of the *Variablize-a-Subtree* transformation. Since VarSQL requires the pattern of a rule to be a valid full or partial SQL query, we cannot variablize an entire clause. For example, in the fourth example in Figure 8, if we variablize the `SELECT <a>` subtree as a new variable <y>, the transformed pattern "`<y> FROM <t> WHERE ...`" is not valid SQL syntax. Thus, we introduce the *Drop-a-Branch* transformation, which removes a common branch in a rule's pattern and replacement. In this way, we gradually remove the irrelevant context of a rule's pattern from the top to the bottom of its AST.

6 SEARCHING FOR HIGH-QUALITY RULES

To solve the rewriting-rule suggestion problem defined in Definition 5.2, we defined a set of transformations in Section 5.2 to generalize the initial rewriting pairs to more general rewriting rules. However, the candidate sets of generalized rules that can cover the initial rewriting examples may be large. It can be computationally expensive to search all possible sets to compute an optimal solution. To solve the problem, we adopt a heuristic-based strategy to expand the candidate-rule set greedily. In this section, we first present the greedy searching framework, then propose several heuristics to further reduce the search overhead.

6.1 A Greedy Searching Framework

We develop a method to search for rules, as shown in Algorithm 1. We start with the original rewriting pairs as a basic solution, and treat each query pair as a rewriting rule (line 1). We iteratively replace rules in the solution with a more general rule that reduces the total description length the most. In each iteration, we first explore a set of candidate rules by applying transformations to the rules in the current solution (line 3). We say a rule x *covers* another rule y if x's pattern matches y's pattern and x can rewrite y's pattern to y's replacement. For each candidate rule, we compute the reduction of the total length if we use it to replace its covered rules in the solution (lines 4-7). We then choose the rule that has the maximum reduction (line 8) and replace its covered rules with the new rule (line 12). We stop the iteration if there is no more reduction (line 9). In this case, we return the current solution (line 10).

Algorithm 1: A greedy algorithm for suggesting rules

```
Input: A set of rewriting pairs
       $Q = \{(Q_1, Q'_1), \dots, (Q_n, Q'_n)\}$
       A set of transformations $\mathcal{T} = \{T_1, T_2, \dots, T_m\}$
       A description length function $\mathcal{L}$ on a rule
Output: A set of rewriting rules $\mathcal{R}$
1  $\mathcal{R} \leftarrow Q$
2  while True do
3      $C \leftarrow$ Explore_Candidates($\mathcal{R}, \mathcal{T}$)
4      for $c \in C$ do
           // find rules that can be replaced by c
5          $\mathcal{R}_c \leftarrow \{R_i \in \mathcal{R} \mid R_i$ is covered by $c\}$
           // compute the length reduction if c replaces $\mathcal{R}_c$
6          $\Delta\mathcal{L}_c \leftarrow \sum_{R_i \in \mathcal{R}_c} \mathcal{L}(R_i) - \mathcal{L}(c)$
7      end
       // choose a candidate rule with the largest length reduction
8      $\hat{c} \leftarrow \arg\max_{c \in C} \Delta\mathcal{L}_c$
       // stop when there is no more reduction
9      if $\Delta\mathcal{L}_{\hat{c}} \leq 0$ then
10         return $\mathcal{R}$
11     end
       // update the result set
12     $\mathcal{R} \leftarrow \mathcal{R} - \mathcal{R}_{\hat{c}} + \hat{c}$
13 end
```

The algorithm follows the hill-climbing paradigm [44], where in each iteration, it explores a set of candidate rules to consider as the possible next directions. The exploration of candidates is implemented in the *Explore_Candidates($\mathcal{R}$,$\mathcal{T}$)* procedure, and the decision of which set of candidates to explore can affect how easily the algorithm is stuck at a local optimum. Ideally, the explored candidates should include all possible rules transformed from the current rule set. However, the size of the transformed rules can be large. Thus, we need to consider the trade-off between the exploration size and the probability of trapping in a local optimum. We discuss different methods in the following.

A naive candidate-exploration method. A naive method is to parameterize the number of hops when we transform the rules in the given rule set. Starting from a base rule, we can transform it into different child rules by applying different transformations. We call a child rule a "1-hop rule" if it is obtained from the base rule by applying one transformation. Similarly, a rule is a "k-hop rule" if it is obtained after applying k transformations on the base rule one by one. The parameter k decides the exploration overhead of the searching framework. We can increase k to allow the algorithm to look ahead before settling down at a local optimum at a higher computational cost. We call this method "k-hop-neighbor exploration" (KHN for short).

This method has two problems. One is that it is hard to decide the k value. A k value may be good for some input examples but can be bad for others. The second problem is that a fixed k value for all base rules ignores their different amounts of potential to discover a high-quality rule. To solve these two problems, we propose an adaptive exploration method next.

6.2 Exploring Candidate Rules Adaptively

In this subsection, we discuss how to explore candidates in an adaptive way by considering the different amounts of potential of transforming different base rules to discover a high-quality general rule. The goal is to explore more promising candidate rules first to fill a fixed size of the candidate set.

Algorithm 2: m-promising-neighbor exploration

```
Input: A set of rewriting rules $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$
       A set of transformations $\mathcal{T} = \{T_1, T_2, \dots, T_m\}$
       A function $\mathcal{P}$ that measures a rule's promisingness
       score
       A parameter $m$ that limits the output size
Output: A set of candidate rewriting rules $C$
1  $C \leftarrow \mathcal{R}$
2  while $|C| < m$ do
       // choose the most promising candidate rule
3      $\hat{c} \leftarrow \arg\max_{c \in C} \mathcal{P}(c)$
       // replace it with its 1-hop transformed child rules
4      for $T_i \in \mathcal{T}$ do
5          $T_i(\hat{c}) \leftarrow$ {all possible child rules by applying $T_i$ to $\hat{c}$}
6          $C \leftarrow C \cup T_i(\hat{c})$
7      end
8      $C \leftarrow C - \hat{c}$
9  end
10 return $C$
```

***m*-promising neighbors.** Its main idea is that instead of exploring neighbors a fixed number of hops away from the current rule set, we explore a fixed number (denoted as m) of neighbors that can reduce the total length of the rule set the most. The value m directly decides the computation overhead of the rule-suggestion algorithm. We can decide its value by considering the running time (e.g., 2 seconds) allowed to run the algorithm and the hardware resources we have. To find the m neighbors, we explore the given base rule set iteratively. In each iteration, we choose a rule that is most promising to be transformed into a more general rule that reduces the total length the most. In this way, we can generate a set of candidate rules with different numbers of hops transformed from different base rules in the given rule set.

Algorithm 2 shows the pseudo-code of the method of m-promising-neighbor exploration (MPN for short). For a given function $\mathcal{P}$ that measures a rule's *promisingness score*, the algorithm starts from the initial rule set, chooses one rule with the highest score, replaces it with all its 1-hop transformed rules in the candidate rule set, and stops until the rule set reaches the given size m.

Measuring the promisingness score of a rule. We consider three signals to measure a rule's promisingness score. One is the total length of those base rules that can be covered if we transform a candidate rule into a more general form. Another is the number of transformations needed to apply to a candidate rule if we want it to cover more base rules in the rule set, which measures how far we can reach a more general rule starting from the current rule. The third signal is the length of a candidate rule.

Formally, given a set of base rewriting rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and a candidate rule c, rule c's promisingness score $\mathcal{P}(c)$ is computed as follows. For each $R_i \in \mathcal{R}$, we compute a distance $\mathcal{D}(c, R_i)$, i.e., the number of transformations on rule c to cover rule R_i. We will discuss how to compute this value shortly. Let $\mathcal{L}$ be the given description length function. The promisingness score of rule c is:

$$\mathcal{P}(c) = \sum_{i=1}^{n} \frac{\mathcal{L}(R_i)}{\mathcal{D}(c, R_i)} + \frac{1}{\mathcal{L}(c)}.$$

If a rule can be generalized with fewer transformations to cover longer base rules and its own length is shorter, it should have a higher promisingness score. We now describe how to compute the distance $\mathcal{D}(c, R_i)$ of transforming rule c to cover the base rule R_i. We count the number of transformations on c to produce a more general form c' to cover rule R_i. A rule c' covers rule R_i if the pattern of c' matches R_i's pattern, and we can rewrite it to R_i's replacement. Therefore, we can run the pattern-matching process of c on R_i similar to that of a rule on a query. The only difference is that when we find any mismatching part, instead of immediately returning false, we compute the number of transformations needed for the mismatching part in c to match that in R_i.

6.3 Including Query Cost in Rule Quality

To this end, we use MDL as a metric for rewriting rules' quality. We now show how to include the effectiveness in improving the performance of a historical workload $\mathcal{W}$ to measure the rules' quality. For a given candidate rewriting rule set $\mathcal{R}$ (in Algorithm 1), we can obtain a set $\mathcal{W}_{\mathcal{R}}$ of rewritten queries by rewriting $\mathcal{W}$ using the rules in $\mathcal{R}$. Suppose we know the cost of queries to the target database. We can obtain the total cost of all rewritten queries in $\mathcal{W}_{\mathcal{R}}$, denoted as $C(\mathcal{W}_{\mathcal{R}})$. When we evaluate the benefit of replacing a few rules in $\mathcal{R}_c$ with a candidate rule c (line 6), we compute the reduction of query cost when using the new rule set to rewrite $\mathcal{W}$, denoted as ΔC_c. Then, we compute a weighted sum of both the reduction of description length and the reduction of query cost as the total benefit for the candidate rule c as

$$Benefit_c \leftarrow \beta \times \frac{\Delta \mathcal{L}_c}{\mathcal{L}_{\mathcal{R}}} + (1 - \beta) \times \frac{\Delta C_c}{C(\mathcal{W}_{\mathcal{R}})},$$

where β is a parameter to tune the balance between the importance of the description length and performance improvement of the rewriting rules. We replace the original $\Delta \mathcal{L}_c$ with the new $Benefit_c$ at lines 6, 8, and 9, and extend Algorithm 1 to include the effectiveness in improving workload performance to measure the quality of rewriting rules. Similarly, we also include the new benefit value when computing the promisingness score of a rule in Algorithm 2.

7 EXPERIMENTS

We conducted experiments to evaluate QueryBooster regarding three aspects: formulating rules using the VarSQL rule language, suggesting rules from user-given examples, and the end-to-end performance using QueryBooster to rewrite queries. In particular, we want to answer the following questions: (1) How easy is it for SQL developers to use the VarSQL language to formulate query rewriting rules? (2) What is the expressive power of VarSQL? (3) Are the transformations defined in QueryBooster enough to generate general rules from example pairs? (4) How do different search strategies perform in terms of running time and rule qualities? (5) How much benefit can QueryBooster provide on the end-to-end query performance with the human-centered query rewritings?

7.1 Setup

Workloads. We used four workloads as shown in Table 3. Each workload had a set of SQL rewriting pairs, and each pair consisted of an original query and a rewritten query. Each rewritten query was equivalent to and usually outperformed its original query. The WeTune workload included 245 pairs of SQL queries published in the appendix table in the paper [54]. They collected those original queries from 20 open source applications on GitHub and generated the rewritten queries by applying their machine-discovered rewriting rules. The Calcite workload comprised 232 rewriting pairs of SQL queries designed for the Apache Calcite test suite [8].

Table 3: Workloads used in the experiments.

Id	Workload	# of query pairs
1	Calcite	232
2	WeTune	245
3	Tableau+TPC-H	20
4	Tableau+Twitter	14 (Postgres) + 6 (MySQL)
5	Superset+Twitter	5 (MySQL)

To consider the real-world use cases where business intelligence (BI) users do interactive analysis on their data residing in a database, we created three more workloads using Tableau [51] and Apache Superset [3] on top of PostgreSQL and MySQL. The "Tableau + TPC-H" workload included 20 rewriting pairs of SQL queries, which corresponded to the top 20 queries in the TPC-H benchmark [53]. We first inserted a 10GB TPC-H synthesized dataset into a PostgreSQL database (indexes were created using Dexter [14]), then used Tableau Desktop software to connect to the PostgreSQL database in its live mode. For each query in the TPC-H benchmark, we manually built a Tableau visualization workbook that could answer the corresponding business question, then collected the backend SQL query generated from Tableau for the workbook. We then analyzed the Tableau-formulated SQL query and came up with a rewritten query with a better performance. Similarly, we generated the "Tableau + Twitter" and "Superset + Twitter" workloads by building visualization dashboards using Tableau and Apache Superset to analyze 30 million tweets on their textual, temporal, and geospatial dimensions on top of both Postgres and MySQL databases. In the workloads, 14 pairs of queries were generated on top of PostgreSQL, and 11 pairs were generated on top of MySQL.

Testbed. We implemented the QueryBooster system using Python 3.9 and used the "mo-sql-parsing" package [26] as the SQL parser. All experiments were run on a MacBook Pro 2017 model with a 2.3GHz Intel Core i5 CPU, 8GB DDR3 RAM, and 256GB SSD. The Tableau Desktop software version was 2021.4, and the Apache Superset version was May 2023. The PostgreSQL software version was 14, and the MySQL software version was 8.0.

Description length function. To evaluate the performance of the rule-suggestion algorithms, we implemented a description length function designed for rules rewritten in VarSQL. We followed the design principles proposed in [40]. The main idea was that each rule had a constant basic length, and the more variables it had, the larger its description length should be. In this case, the description length metric made sure that high-quality rules could match as many given examples as possible, but they were not over-generalized to match unseen queries. We computed the description length $\mathcal{L}$ of a particular rule r as the following. Let W be the constant basic length of any rule, W_E be the weight of an element-variable, and W_S be the weight of a set-variable. We used three counters in the given rule. We counted the number of element-variables in the rule as C_E and the number of set-variables as C_S. In addition, we counted the number of non-variable elements in the rule as C_O, where non-variable elements included keywords, values, table names, column names, etc. In the end, we computed the length $\mathcal{L}$ of rule r as

$$\mathcal{L}(r) = W + (W_E \times C_E + W_S \times C_S)/C_O.$$

7.2 A User Study to Evaluate Rule Languages

Table 4: User profiles in the user study.

Background	Faculty	Staff	Software Engineers	Ph.D. students	M.S. students
% of users	4.5%	4.5%	4.5%	72.7%	13.6%

We conducted a user study to evaluate how easy it was for SQL users to use VarSQL to formulate rewriting rules. Besides VarSQL, we considered two other languages for comparison. One was regular expression [55], and we used its C Sharp implementation provided by regex101 [42]. The other was the internal rule language used by WeTune [54], and we used its own implementation provided by its demo website WeRewriter [22]. We selected three rewriting pairs of SQL queries from two workloads on two databases. One pair was from the WeTune workload, and the other two pairs were from the "Tableau + Twitter" workload on both PostgreSQL and MySQL. For each rewriting pair, we showed the original and rewritten queries to the user, along with three rewriting rules defined in the three languages that could achieve the same rewriting. We asked the user to "*select one of the three rules that you think is the easiest to understand.*" In the questions, we randomized the orders of the rule languages and hid their names to make the comparison fair.

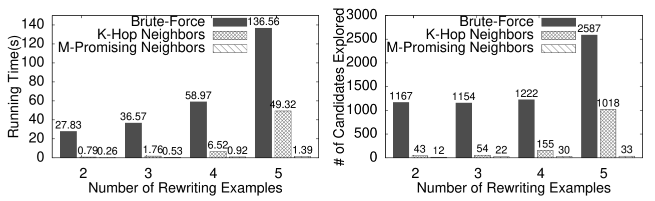
We invited 22 users who were familiar with SQL and with different backgrounds. The profiles of users are shown in Table 4, and the results are summarized in Table 5. Among all the rewriting pairs, more than 80% of users selected the rule formulated in VarSQL as the easiest to understand, and it outperformed the other two languages significantly. The user study results show that VarSQL is an easy-to-use language and was preferred by SQL users.

Table 5: Results in the user study (% of users selected the rule language as the easiest to understand).

Pair Id	1	2	3
Workload	Twitter(Postgres)	Twitter(MySQL)	WeTune(Q91)
% of Regex	13.6%	4.5%	0%
% of WeTune	0%	13.6%	13.6%
% of VarSQL	**86.4%**	**81.8%**	**86.4%**

7.3 Comparison of Rule-Searching Strategies

We evaluated the performance of different searching strategies in the rule-suggestion searching framework. We compared the three strategies discussed in Section 6. The first was "Brute-Force" ("BF" for short), which explored all possible rules that were transformed from the current rule set in the *Explore_Candidates* procedure. The second was the "k-hop-neighbor exploration" ("KHN" for short), where we explored the neighbors of a fixed number (k) of hops away from the base rules for each iteration's consideration. The last was the adaptive exploration method, "m-promising-neighbor exploration" ("MPN" for short), where we explored a fixed number (m) of neighbors that were the most promising to finally reduce the total description length of the resulting rule set. We used the "Tableau + Twitter" workload and varied the number of rewriting examples as the input to the searching algorithms. For each input set of examples, we first ran the BF method to get a high-quality set of suggested rules as the benchmark. We then ran the KHN and MPN methods and made sure they both output the same set of suggested rules as the BF method by gradually increasing the k and m parameters. In this way, we ensured the fairness of the comparison between different methods.



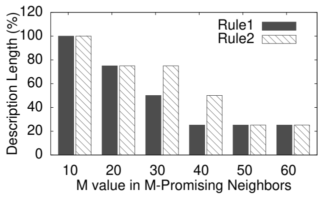
(a) Running time. **(b) Total # of candidates explored.**

Figure 9: Comparison of different candidate exploration methods to suggest the same set of rules on the "Tableau + Twitter" workload.

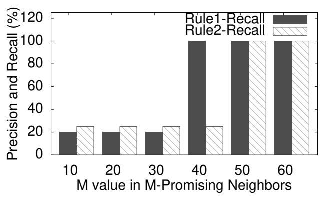
As shown in Figure 9a, as the number of input examples increased, the running time of the brute-force method increased sub-exponentially. The reason was for each example added to the input set, the number of candidate rules generated from the new example was exponential to its number of elements in the original query. Compared to the brute-force method, the KHN method had significantly less running time since it only explored a small set of candidate rules during the exploration phase. However, its running time still went up to 50 seconds for 5 input examples. The reason was that to reach the high-quality rules, the KHN method had to tune its k value to 4, and the number of explored rules increased exponentially with the increase of the k value. In comparison, the MPN method outperformed both other methods significantly, and the running time increased linearly as the input set size increased. These results are consistent with those shown in Figure 9b, and both figures illustrate the correlation between the running time and the number of candidates explored in the searching framework.

7.4 Effect of m in m-promising Neighbors

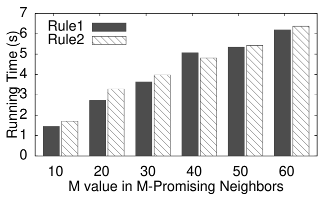
We evaluated the effect of the m value in the m-promising-neighbor searching strategy on the WeTune workload. We randomly chose 30 rewriting pairs within the first three applications in the workload as the testing set. We then chose the top two frequent rewriting patterns and named them as "*Rule1*" and "*Rule2*". Among the 30 pairs, there were 5 pairs matching *Rule1* and 4 pairs matching *Rule2*. For each rule, we used one matching pair as the seed and manually generated 4 rewriting examples as the input examples for the rule-suggestion algorithm. We ran the algorithm using the m-promising-neighbor strategy with different m values. We measured the total description length of the output rule set and the result is shown in Figure 10a. It shows that for both rules' input example sets when the m value increased, the output of the rule-suggestion algorithm converged to the optimal rule set with the minimum description length. Referring to the corresponding running time shown in Figure 10c, it only took about 5 to 6 seconds for the algorithm to output the optimal rule set. Figure 10d also shows the numbers of candidates explored for different m values.



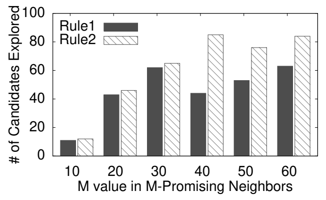
(a) Description Length (% of the raw examples).



(b) Precision and Recall on unseen pairs.



(c) Running Time (s).



(d) Total # of candidates explored.

Figure 10: Effect of the m value in the m-promising-neighbor searching strategy on the WeTune workload.

We evaluated the output rule set from the rule-suggestion algorithm on the unseen 30 testing rewriting pairs in the workload. We measured both the precision and recall computed as follows. Suppose the rule set rewrote x unseen pairs of queries, among which $x1$ pairs satisfied the intent of the user. Then the precision is $\frac{x1}{x}$. Suppose the user wanted y pairs of queries in the testing set to be successfully rewritten, and the rule set only rewrote $y1$ out of y. Then the recall is $\frac{y1}{y}$. The result is shown in Figure 10b. The precision was always 100% (omitted in the figure) because the design of the description length function enforced that the rules were never over-generalized. And the recall was initially low for a small m value because the output rules were very specific to the input examples, and the output rules were not optimal yet. As the m value increased to 50 or more, the algorithm started to output the optimal suggested rules that could cover unseen query pairs with similar patterns, which led to a 100% recall in the end.

7.5 End-to-End Query Time Using QueryBooster

We evaluated the end-to-end query time (the time between the frontend sending the SQL query to and receiving the result from the database) using QueryBooster to rewrite queries in the "Tableau + TPC-H" and "Tableau + Twitter" workloads on PostgreSQL and the "Superset + Twitter" workload on MySQL. For each query in the workload, besides the running time of the original query formulated by Tableau or Superset on PostgreSQL or MySQL, we also collected the running time of two rewritten queries using different rewriting rules. One rewritten query (noted as "*Rewritten Query (WeTune Rules)*") was obtained from the WeRewriter [22] system, which used rewriting rules automatically discovered by WeTune. The other rewritten query (noted as "*Rewritten Query (Human Rules)*") was obtained from QueryBooster using human-crafted rewriting rules based on manual analysis of the original query and its plan.

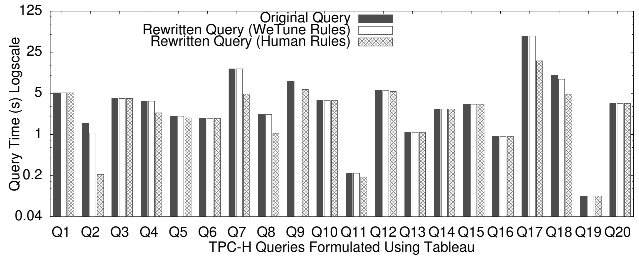
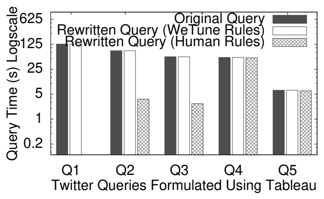


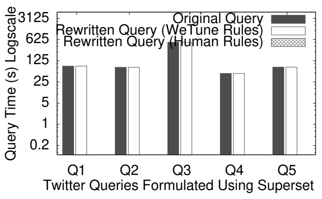
Figure 11: End-to-end query time using QueryBooster to rewrite queries with WeTune-generated rules and human-crafted rules on "Tableau + TPC-H" workload compared to original query time in PostgreSQL.

Figure 11 shows the result for the workload of "Tableau + TPC-H" on Postgres. Among the 20 queries, only two rewritten queries ($Q2$ and $Q18$) using the WeTune-generated rules could reduce the query time. At the same time, using human-crafted rewriting rules, QueryBooster reduced 10 queries' running time, which comprised 50% of all the queries. Within the 10 rewritten queries using human-crafted rules, 70% of them reduced the original queries' running time significantly (by more than 25%). For example, $Q2$ was reduced by 86% (1.555s to 0.207s) and $Q17$ was reduced by 61% (47.046s to 17.802s). Note that in the 10 queries optimized using human-crafted rules, 7 of them used statement-level reshaping rules such as "join-to-exists", "remove-subquery", etc., and 3 of them used hints such as "force-join-order".

Figure 12a and 12b show the results for the workloads of "Tableau + Twitter" and "Superset + Twitter" on MySQL. The result of "Tableau + Twitter" on PostgreSQL was similar to MySQL, thus not shown. For the 5 Tableau queries on MySQL, the human-crafted rewriting rules were mainly predicate-level removing unnecessary `ADDDATE` calculation, as discussed in Section 2. For the 5 Superset queries on MySQL, the human-crafted rewriting rules were mainly translating the textual filtering condition from a `LIKE` predicate to a full-text search predicate since MySQL does not support any index-scan for `LIKE` predicate but does for full-text search. For example, in one query, the human-crafted rule rewrote the predicate "`text LIKE '%stopasian hate%'`" to "`MATCH(text) AGAINST ('stopasianhate stopasian hatecrime stopasianhatecrimes')`, which was equivalent only for this particular dataset because all substring "stopasianhate" matched records could be matched using the three full-text keywords: "stopasianhate", "stopasianhatecrime", and "stopasianhatecrimes". As shown in Figure 12b, all 5 queries were accelerated by 100+ times (e.g., 83s to 0.8s) due to this human-crafted rewriting rule, which shows the importance of the proposed human-centered query rewriting approach.



(a) "Tableau + Twitter".

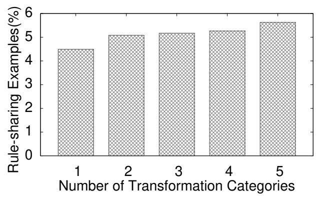


(b) "Superset + Twitter".

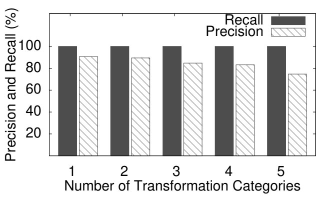
Figure 12: End-to-end query time using QueryBooster **to rewrite queries with WeTune-generated rules and human-crafted rules compared to original query time in MySQL.**

7.6 Generality of Rule Transformations

To evaluate the generality (covering more rewriting examples) of rule transformations, we used 178 rewriting pairs in the Calcite workload. We applied the transformations defined in Section 5.2 to each example iteratively to generate more general rules. We divided the transformations into 5 categories: "*variablize-a-table*", "*variablize-a-column*", "*variablize-a-value*", "*variablize-a-subtree*", and "*merge-variables*", and gradually applied more categories to generalize the rules. We collected the percentage of examples that were rewritten by rules generated by other examples and named it "*sharing-rule examples (%)*". We also collected the precision and recall of using the generalized rules to rewrite the original queries.



(a) Rule sharing percentage.



(b) Precision and Recall.

Figure 13: The generality of rules generalized from the Calcite examples using different sets of transformations.

Figure 13a shows that with more transformation categories used in generating rules, more examples shared rules, meaning the generated rules were more general. Figure 13b shows that with more transformations used to generalize rules, the recall remained 100% because more general rules could always match the seed examples. However, the rewriting precision went down. The reason was that more transformations resulted in over-generalized rules that matched examples they should not match. This result also motivated our consideration of using MDL to prevent over-generalization.

7.7 Effect of Different Rule Quality Metrics

We also evaluated the effect of using different importance weights β in the *benefit* value (defined in Section 6.3) using the "Tableau + Twitter" workload. We used four query pairs as input examples for the rule-suggesting framework and another five queries as a historical workload to compute the benefit value in Algorithm 1. We varied β from 1.0 (i.e., only considering the MDL as rule quality) to 0.0 (i.e., only considering query cost as rule quality). For each β value, we first ran the rule-suggesting framework with the "MPN" strategy to obtain the suggested rules. We then evaluated the suggested rules based on three metrics. We used the rules to rewrite the five queries in the workload and collected the query "cost reduction" by comparing the rewritten queries' cost and the original queries' cost. We treated the suggested rules with $\beta = 1.0$ as the baseline and then computed the "description length increase" and rule-suggesting algorithm "running time increase" for the rules suggested by other β values.

The results are shown in Figure 14. When β was 1.0, the suggested rules only reduced the query cost by 18%. When β decreased to 7.5, the suggested rules reduced the query cost by 90%. However, the cost was both the description length of rules and the running time of the rule-suggesting algorithm increased by 40%. The cost increased when β further decreased. When β was 0.0, which means the algorithm did not consider the description length at all, the total description length of suggested rules increased by 280%.

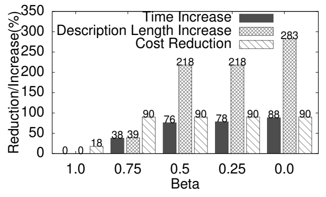


Figure 14: Effect of different β values.

Remarks: The user study shows that more than 80% SQL users preferred using the VarSQL rule language to formulate rewriting rules. QueryBooster suggested high-quality (high precision and recall and low description length) rules from user-given examples quickly ($\leq$ 5s) on different workloads. Compared to existing query rewriting solutions with machine-discovered rewriting rules, using QueryBooster with human-crafted rewriting rules improved the performance of 50% TPC-H queries by up to 86%.

8 CONCLUSIONS

In this paper, we proposed QueryBooster, a middleware service for human-centered query rewriting. We developed a novel expressive rule language (VarSQL) for users to formulate rewriting rules easily. We designed a rule-suggestion framework that automatically suggests high-quality rewriting rules from user-given examples. A user study and experiments on various workloads show the benefit of using VarSQL to formulate rewriting rules, the effectiveness of the rule-suggestion framework, and the significant advantages of using QueryBooster to improve the end-to-end query performance.

REFERENCES

[1] APACHE AsterixDB [n.d.]. http://asterixdb.apache.org. last accessed: 7-19-2023.

[2] Apache Calcite [n.d.]. https://calcite.apache.org/. last accessed: 7-19-2023.

[3] Apache Superset(incubating) - Apache Superset documentation. 2018. https://superset.incubator.apache.org/. last accessed: 7-19-2023.

[4] ApexSQL: SQL execution plan viewing and analysis [n.d.]. https://www.apexsql.com/sql-tools-plan.aspx. last accessed: 7-19-2023.

[5] Qiushi Bai, Sadeem Alsudais, and Chen Li. 2022. Demo of VisBooster: Accelerating Tableau Live Mode Queries Up to 100 Times Faster. In *Proceedings of the Workshops of the EDBT/ICDT 2022 Joint Conference, Edinburgh, UK, March 29, 2022 (CEUR Workshop Proceedings)*, Maya Ramanath and Themis Palpanas (Eds.), Vol. 3135. CEUR-WS.org. http://ceur-ws.org/Vol-3135/bigvis_short5.pdf

[6] Qiushi Bai, Sadeem Alsudais, Chen Li, and Shuang Zhao. 2023. Maliva: Using Machine Learning to Rewrite Visualization Queries Under Time Constraints. In *Proceedings 26th International Conference on Extending Database Technology, EDBT 2023, Ioannina, Greece, March 28-31, 2023*, Julia Stoyanovich, Jens Teubner, Nikos Mamoulis, Evaggelia Pitoura, and Jan Mühlig (Eds.). OpenProceedings.org, 157–170. https://doi.org/10.48786/edbt.2023.13

[7] Falk Brauer, Robert Rieger, Adrian Mocan, and Wojciech M. Barczynski. 2011. Enabling information extraction by inference of regular expressions from sample entities. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, Craig Macdonald, Iadh Ounis, and Ian Ruthven (Eds.). ACM, 1285–1294. https://doi.org/10.1145/2063576.2063763

[8] Calcite Test Suite [n.d.]. https://github.com/georgia-tech-db/spes/blob/main/testData/calcite_tests.json. last accessed: 7-19-2023.

[9] Mitch Cherniack and Stanley B. Zdonik. 1996. Rule Languages and Internal Algebras for Rule-Based Optimizers. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 401–412. https://doi.org/10.1145/233269.233356

[10] 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 (2018), 1482–1495. https://doi.org/10.14778/3236187.3236200

[11] Comby is a tool for searching and changing code structure [n.d.]. https://comby.dev/. last accessed: 7-19-2023.

[12] Guilherme Damasio, Vincent Corvinelli, Parke Godfrey, Piotr Mierzejewski, Alexandar Mihaylov, Jaroslaw Szlichta, and Calisto Zuzarte. 2019. Guided automated learning for query workload re-optimization. *Proc. VLDB Endow.* 12, 12 (2019), 2010–2021. https://doi.org/10.14778/3352063.3352120

[13] Dinesh Das and Don S. Batory. 1995. Praire: A Rule Specification Framework for Query Optimizers. In *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, Philip S. Yu and Arbee L. P. Chen (Eds.). IEEE Computer Society, 201–210. https://doi.org/10.1109/ICDE.1995.380391

[14] Dexter: The automatic indexer for Postgres [n.d.]. https://github.com/ankane/dexter. last accessed: 7-19-2023.

[15] EverSQL: Automatic SQL Query Optimization for MySQL and PostgreSQL [n.d.]. https://www.eversql.com/. last accessed: 7-19-2023.

[16] Béatrice Finance and Georges Gardarin. 1991. A Rule-Based Query Rewriter in an Extensible DBMS. In *Proceedings of the Seventh International Conference on Data Engineering, April 8-12, 1991, Kobe, Japan*. IEEE Computer Society, 248–256. https://doi.org/10.1109/ICDE.1991.131472

[17] Minos N. Garofalakis, Aristides Gionis, Rajeev Rastogi, S. Seshadri, and Kyuseok Shim. 2003. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data Min. Knowl. Discov.* 7, 1 (2003), 23–56. https://doi.org/10.1023/A:1021560618289

[18] Goetz Graefe and David J. DeWitt. 1987. The EXODUS Optimizer Generator. In *Proceedings of the Association for Computing Machinery Special Interest Group on Management of Data 1987 Annual Conference, San Francisco, CA, USA, May 27-29, 1987*, Umeshwar Dayal and Irving L. Traiger (Eds.). ACM Press, 160–172. https://doi.org/10.1145/38713.38734

[19] Yeye He, Xu Chu, Kris Ganjam, Yudian Zheng, Vivek R. Narasayya, and Surajit Chaudhuri. 2018. Transform-Data-by-Example (TDE): An Extensible Search Engine for Data Transformations. *Proc. VLDB Endow.* 11, 10 (2018), 1165–1177. https://doi.org/10.14778/3231751.3231766

[20] Denis Hirn and Torsten Grust. 2019. PgCuckoo: Laying Plan Eggs in PostgreSQL's Nest. In *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, Peter A. Boncz, Stefan Manegold, Anastasia Ailamaki, Amol Deshpande, and Tim Kraska (Eds.). ACM, 1929–1932. https://doi.org/10.1145/3299869.3320211

[21] IBM DB2 11.5: Query rewriting methods and examples [n.d.]. https://www.ibm.com/docs/en/db2/11.5?topic=process-query-rewriting-methods-examples. last accessed: 7-19-2023.

[22] Jinyuan Zhang and Yicun Yang. 2023. https://ipads.se.sjtu.edu.cn/werewriter-demo/home. last accessed: 7-19-2023.

[23] Keebo: Data Learning and Warehouse Optimization [n.d.]. Keebo: Data Learning and Warehouse Optimization. https://keebo.ai/.

[24] Kapil Khurana and Jayant R. Haritsa. 2021. Shedding Light on Opaque Application Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 912–924. https://doi.org/10.1145/3448016.3457252

[25] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[26] Kyle Lahnakoski. 2023. https://github.com/klahnakoski/mo-sql-parsing. last accessed: 7-19-2023.

[27] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1275–1288. https://doi.org/10.1145/3448016.3452838

[28] MongoDB 5.0: Query documents [n.d.]. https://www.mongodb.com/docs/manual/tutorial/query-documents/. last accessed: 7-19-2023.

[29] MySQL 8.0: 5.6.4.2 Using the Rewriter Query Rewrite Plugin [n.d.]. https://dev.mysql.com/doc/refman/8.0/en/rewriter-query-rewrite-plugin-usage.html. last accessed: 7-19-2023.

[30] Oracle Free Use Terms and Conditions [n.d.]. https://www.oracle.com/downloads/licenses/oracle-free-license.html. last accessed: 7-19-2023.

[31] Oracle R19: 11 Basic Query Rewrite for Materialized Views [n.d.]. https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/basic-query-rewrite-materialized-views.html. last accessed: 7-19-2023.

[32] Lionel Parreaux, Antoine Voizard, Amir Shaikhha, and Christoph E. Koch. 2018. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.* 2, POPL (2018), 13:1–13:33. https://doi.org/10.1145/3158101

[33] Hamid Pirahesh, Joseph M. Hellerstein, and Waqar Hasan. 1992. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 2-5, 1992*, Michael Stonebraker (Ed.). ACM Press, 39–48. https://doi.org/10.1145/130283.130294

[34] Hamid Pirahesh, T. Y. Cliff Leung, and Waqar Hasan. 1997. A Rule Engine for Query Transformation in Starburst and IBM DB2 C/S DBMS. In *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, W. A. Gray and Per-Åke Larson (Eds.). IEEE Computer Society, 391–400. https://doi.org/10.1109/ICDE.1997.581945

[35] PostgreSQL 14: CREATE RULE — define a new rewrite rule [n.d.]. https://www.postgresql.org/docs/14/sql-createrule.html. last accessed: 7-19-2023.

[36] PostgreSQL 14 Documentation: 41.2. Views and the Rule System [n.d.]. https://www.postgresql.org/docs/current/rules-views.html. last accessed: 7-19-2023.

[37] PostgreSQL 14: Trigram index [n.d.]. https://www.postgresql.org/docs/current/pgtrgm.html. last accessed: 7-19-2023.

[38] Query Performance Insight for Azure SQL Database [n.d.]. https://docs.microsoft.com/en-us/azure/azure-sql/database/query-performance-insight-use?view=azuresql. last accessed: 7-19-2023.

[39] Query Rewrite and Optimization [n.d.]. https://docs.teradata.com/r/8mHBBLGP88~HK9Auie2QvQ/4PC2qalhztpNrpq9R~zpDw. last accessed: 7-19-2023.

[40] Vijayshankar Raman and Joseph M. Hellerstein. 2001. Potter's Wheel: An Interactive Data Cleaning System. In *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, Peter M. G. Apers, Paolo Atzeni, Stefano Ceri, Stefano Paraboschi, Kotagiri Ramamohanarao, and Richard T. Snodgrass (Eds.). Morgan Kaufmann, 381–390. http://www.vldb.org/conf/2001/P381.pdf

[41] Re: How to use index in strpos function [n.d.]. https://www.postgresql.org/message-id/046801c96b06%242cb14280%248613c780%24%40r%40sbcglobal.net. last accessed: 7-19-2023.

[42] regular expressions 101 [n.d.]. https://regex101.com/. last accessed: 7-19-2023.

[43] Jorma Rissanen. 1978. Modeling by shortest data description. *Autom.* 14, 5 (1978), 465–471. https://doi.org/10.1016/0005-1098(78)90005-5

[44] Stuart Russell and Peter Norvig. 2020. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson. http://aima.cs.berkeley.edu/

[45] SAP HANA Performance Guide for Developers [n.d.]. https://help.sap.com/doc/05b8cb60dfd94c82b86828ee77f7e0d9/2.0.04/en-US/SAP_HANA_Performance_Developer_Guide_en.pdf. last accessed: 7-19-2023.

[46] Scala: Quasiquotes Introduction [n.d.]. https://docs.scala-lang.org/overviews/quasiquotes/intro.html. last accessed: 7-19-2023.

[47] Edward Sciore and John Sieg Jr. 1990. A Modular Query Optimizer Generator. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*. IEEE Computer Society, 146–153. https://doi.org/10.1109/ICDE.1990.113464

[48] Snowflake documentation [n.d.]. https://docs.snowflake.com/en/index.html. last accessed: 7-19-2023.

[49] Software is fragile [n.d.]. https://www.softwareheritage.org/mission/software-is-fragile/. last accessed: 7-19-2023.
[50] SQL Server 2019: SQL Server technical documentation [n.d.]. https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15. last accessed: 7-19-2023.
[51] Tableau [n.d.]. https://www.tableau.com/. last accessed: 7-19-2023.
[52] Toad: Develop, analyze, and administer databases with Toad [n.d.]. https://www.toadworld.com/products. last accessed: 7-19-2023.
[53] TPC-H Website [n.d.]. http://www.tpc.org/tpch/. last accessed: 7-19-2023.
[54] Zhaoguo Wang, Zhou Zhou, Yicun Yang, Haoran Ding, Gansen Hu, Ding Ding, Chuzhe Tang, Haibo Chen, and Jinyang Li. 2022. WeTune: Automatic Discovery and Verification of Query Rewrite Rules. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 94–107. https://doi.org/10.1145/3514221.3526125
[55] Wiki: Regular expression [n.d.]. https://en.wikipedia.org/wiki/Regular_expression. last accessed: 7-19-2023.
[56] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A Learned Query Rewrite System using Monte Carlo Tree Search. *Proc. VLDB Endow.* 15, 1 (2021), 46–58. https://doi.org/10.14778/3485450.3485456