UNIVERSITY OF CALIFORNIA,
IRVINE

Improving SQL Performance Using Middleware-Based Query Rewriting

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Qiushi Bai

Dissertation Committee:
Professor Chen Li, Chair
Professor Michael J. Carey
Professor Sharad Mehrotra

2023

DEDICATION

To the love of my life,
Xiaofei Zhou,
whose unconditional support, sacrifices, and belief in me
gave me the strength and motivation
to complete this journey.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **ix**

ACKNOWLEDGMENTS **x**

VITA **xi**

ABSTRACT OF THE DISSERTATION **xii**

1 Introduction **1**

2 Using Machine Learning to Rewrite Visualization Queries Under Time Constraints **5**

2.1 Introduction 5

2.1.1 Related Work 9

2.2 Problem Formulation 11

2.3 Maliva: ML-based Query Rewriting 14

2.4 MDP Model for Adding Query Hints 16

2.4.1 MDP Model for Query Rewriting 16

2.4.2 Query Time Estimator (QTE) 22

2.5 Training and Using the MDP Agent 22

2.5.1 Training the MDP Agent 22

2.5.2 Using MDP to Rewrite Queries Online 26

2.6 Approximation Rewriting Options 26

2.6.1 Quality-Aware MDP Model 28

2.6.2 Quality-Aware Query Rewriter 30

2.6.3 Differences between Maliva and Bao 32

2.7 Experiments 32

2.7.1 Setup 33

2.7.2 Performance on Using Query Hints 36

2.7.3 Effect of Rewrite-Option Number 38

2.7.4 Effect of Time Budget 39

2.7.5 Performance on Join Queries 40

2.7.6 Additional Comparison with Bao 41

2.7.7 Unseen Queries and Other Databases 42
2.7.8 Performance of Quality-Aware Rewriting 44
2.7.9 Training Performance 45
2.8 Conclusions 46

3 Supporting Human-Centered Query Rewriting in Middleware 48
3.1 Introduction 48
3.2 Limitations of Existing Solutions 52
3.3 QueryBooster: A Human-Centered Query Rewriting Solution 55
3.4 VarSQL: A Rewriting-Rule Language 58
3.4.1 Suitability of Existing Rule Languages 58
3.4.2 VarSQL: A Novel Rule Language 61
3.4.3 VarSQL-Based Rewriting Engine 66
3.5 Rule Quality and Transformations 69
3.5.1 Quality of Rewriting Rules 69
3.5.2 Transforming Rules to More General Forms 72
3.6 Searching For High-Quality Rules 75
3.6.1 A Greedy Searching Framework 76
3.6.2 Exploring Candidate Rules Adaptively 78
3.6.3 Including Query Cost in Rule Quality 80
3.7 Experiments 81
3.7.1 Setup 81
3.7.2 A User Study to Evaluate Rule Languages 83
3.7.3 Comparison of Rule-Searching Strategies 84
3.7.4 Effect of m in m-promising Neighbors 86
3.7.5 End-to-End Query Time Using QueryBooster 88
3.7.6 Generality of Rule Transformations 90
3.7.7 Effect of Different Rule Quality Metrics 91
3.8 Conclusions 93

4 Supporting Middleware-Based SQL Query Rewriting as a Service 94
4.1 Introduction 94
4.2 QueryBooster System Overview 97
4.3 Demonstration of the User Experience 99
4.3.1 Formulating Rules through a Rule Language 99
4.3.2 Formulating Rules by Providing Examples 102
4.3.3 Suggesting Useful Rules from Other Users 103
4.4 Supporting Management and Sharing Rewritings 104
4.4.1 Informative and Fully-Controllable Rewriting Management 106
4.4.2 Intuitive and Interactive Sharing of Rewriting Knowledge 109
4.5 Minimizing Database Intrusiveness 114
4.5.1 Intrusiveness of SQL Query Rewriting Systems 114
4.5.2 A Survey of Database Connector Licences 117
4.5.3 Profiling Application Queries through Database Connectors 118
4.6 Conclusions 121

5 Conclusions and Future Work **122**

5.1 Conclusions 122

5.2 Future Work 123

Bibliography **126**

LIST OF FIGURES

Page

2.1 Equivalent rewriting option: adding query hints helps the database compute results within a time budget ($500ms$). 6

2.2 Approximation rewriting option: rewriting the query to compute an approximate result within the time constraint. 7

2.3 A original query and a rewritten query. 12

2.4 The Query Rewriter acts like an agent who makes a sequence of decisions to generate a rewritten query (with a total time $\leq 500ms$). At time 0, the agent considers rewritten query RQ_1 due to its low estimation cost (estimated $25ms$, the actual $30ms$ is updated once RQ_1 is explored). After estimating its execution time ($1,300ms$), the agent knows that RQ_1 is not viable since the total time is longer than $500ms$. The estimation of RQ_1 affects the costs for estimating RQ_5 and RQ_7. The agent explores RQ_5 and then RQ_7. With the estimated execution time being $300ms$ and the elapsed time being $150ms$, RQ_7 is decided as a viable rewritten query because the total time ($450ms$) is within $500ms$. 14

2.5 Overview of Maliva. 15

2.6 An MDP state in Maliva. 18

2.7 Transition after estimating execution time of RQ_1. 20

2.8 The quality of RQ compared to Q using a Jaccard-based quality function as an example. 29

2.9 One-stage MDP approach. 30

2.10 Two-stage MDP approach. After running the original agent that considers the 8 query-hint sets defined in Figure 2.4 without approximation rules, we cannot find a viable RQ. We then run the new agent with the quality-aware MDP model that considers all 8 query-hint sets combined with 3 approximation-rule sets (e.g., substituting the `tweets` table with 20%, 40%, or 80% sample tables), resulting in 24 rewritten queries in total. After spending extra time exploring a few RQs, the quality-aware agent chooses RQ_{21} as the final decision. 31

2.11 Performance on the Twitter dataset ($\tau = 500ms$). 37

2.12 Performance on the NYC Taxi dataset ($\tau = 1s$). 37

2.13 Performance on the TPC-H dataset ($\tau = 500ms$). 37

2.14 Performance for 16 ROs on the Twitter dataset ($\tau = 500ms$). 39

2.15 Performance for 1-second time budget on the Twitter dataset. 40

2.16 Performance for join queries (Twitter dataset, $\tau = 500ms$). 41

2.17 Comparison with Bao on the Twitter dataset ($\tau = 500ms$). 42
2.18 Generalization to (a) handle unseen queries and (b) use a commercial database. 43
2.19 Performance of quality-aware rewriting (Twitter, $\tau = 500ms$). 44
2.20 Learning curve and training time on the Twitter dataset. The shaded area is plotted with "mean + standard deviation" as the upper bound and "mean − standard deviation" as the lower bound. 46

3.1 Query lifecycle between Tableau and Postgres. 48
3.2 An example query pair (differences shown in blue). 49
3.3 Query-rewriting solutions for databases (native solutions in brown and third-party solutions in blue). 53
3.4 Query lifecycle of using QueryBooster to rewrite application queries. 56
3.5 Existing rule languages (shown in brown) in the lifecycle of a SQL query. . . 59
3.6 Rewriting rules to replace STRPOS functions with ILIKE predicates in two different languages. 60
3.7 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 Q's elements. 63
3.8 An example rule in VarSQL that removes an unnecessary self-join. 65
3.9 An example of three possible logical plans for a single SQL query which joins tables A, B, and C with a filtering predicate on the z column of C. 67
3.10 Two original queries that both match the pattern of the rule in Figure 3.8. . 68
3.11 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. 70
3.12 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. 73
3.13 A rule graph generated from a given rewriting pair (Q, Q'). The vertices are generalized rules (only showing patterns due to the space limit). The solid edges show one path of generalizing the pair into a general rule. The green tags on the edges illustrate which transformations are applied. 74
3.14 Comparison of different candidate exploration methods to suggest the same set of rules on the "Tableau + Twitter" workload. 85
3.15 Effect of the m value in the m-promising-neighbor searching strategy on the WeTune workload. 87
3.16 End-to-end query time (Log-scale) 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. 88
3.17 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. 90

3.18 The generality of rules generalized from the Calcite examples using different sets of transformations. 91
3.19 Effect of different β values. 92

4.1 QueryBooster overview. 95
4.2 QueryBooster system architecture. 98
4.3 The “Query Logs” page of QueryBooster shows information about queries from the application. 100
4.4 Alice formulates a rule to remove `CAST(··· AS TEXT)` expressions using the VarSQL language [88]. In VarSQL’s syntax, `<x>` is an element-variable that represents a table, column, value, expression, predicate, or sub-query. 100
4.5 QueryBooster shows the rewriting path of a query. 101
4.6 Queries and plans before and after the rewriting of replacing the `STRPOS()>0` predicate with the `ILIKE` predicate. 102
4.7 QueryBooster suggests a rule given a rewriting example. 103
4.8 QueryBooster suggests Bob a rewriting for a query. 104
4.9 QueryBooster suggests a rewriting for Bob’s query by applying Alice’s two rules.105
4.10 The dashboard of rewriting-rule analysis in QueryBooster. 106
4.11 The ER diagram of the database of QueryBooster. 107
4.12 An example of using macro rules to enforce a user-desired rewriting sequence for a query. 108
4.13 QueryBooster suggests rewriting rules interactively. The user can select interesting rules and see the corresponding rewritten query by applying the rule and the estimated or real query time of both the original and rewritten queries. Users can explore different rewriting rules and sequences based on their database and rewriting needs. 112
4.14 The workflow of profiling the database running time of a rewritten query generated by the QueryBooster rewriting service through a PostgreSQL JDBC driver. 119

LIST OF TABLES

Page

2.1 Datasets. 33
2.2 Number of queries in evaluation workloads. 36
2.3 Workloads with 16 rewriting options. 38
2.4 Workloads of queries where less than 50% plans were viable. 41

3.1 Suitability of existing languages for QueryBooster. 61
3.2 Variable definitions in VarSQL. 64
3.3 Workloads used in the experiments. 82
3.4 User profiles in the user study. 83
3.5 Results in the user study (% of users selected the rule language as the easiest to understand). 84

4.1 Intrusiveness levels of systems. 115
4.2 JDBC driver licenses for database vendors. 117

ACKNOWLEDGMENTS

Throughout my six-year journey as a Ph.D. student, I have been fortunate to have the unwavering support and guidance of my advisor, Professor Chen Li. I express my deepest gratitude to him for his invaluable contribution to every aspect of my research, including topic selection, technique development, and paper writing. Professor Li's visionary approach paved the way for the successful QueryBooster project, which later evolved into the Squidster project. His unique perspective and rigorous research methodology have profoundly impacted my intellectual growth, molding me into an independent researcher. I owe my deepest gratitude to him, as without his unwavering support, this dissertation would not have been possible.

I express my sincere gratitude to Professor Michael J. Carey and Professor Sharad Mehrotra for serving on my doctoral committee. Their expertise and insightful feedback have consistently strengthened my work, and I am honored to have had the opportunity to learn from them.

I am immensely grateful to my Master's degree advisor, Professor Ge Yu, and my Bachelor's degree advisor, Professor Yunzhou Zhang, whom I met while at Northeastern University in China. Their helpful guidance and support have played a pivotal role in shaping my academic journey, and without them, I would not have had the opportunity to pursue this path.

I would like to thank Dr. Jianfeng Jia and Dr. Taewoo Kim for their kind assistance during my initial days with the team. Our collaboration on the Cloudberry project enabled me to assimilate the Ph.D. study style at UC Irvine quickly.

I am grateful to all my collaborators who have been a part of my Ph.D. work at UC Irvine, including Sadeem Alsudais, Professor Shuang Zhao, Dr. Liming Dong, Niklas Stoehr, Professor Taiji Chen, Lu He, Yicong Huang, Professor Suellen Hopfer, Dr. Ted Grover, and Professor Gloria Mark. Their contributions have been instrumental in advancing my research.

I also express my gratitude to my colleagues at UC Irvine who have made my Ph.D. journey an enriching experience, including Dayue Bai, Yang Cao, Teyu Chen, Rui Guo, Shanshan Han, Dr. Avinash Kumar, Yue Leng, Xinyuan Lin, Fangqi Liu, Xiaozhen Liu, Chenhan Lyu, Dr. Chen Luo, Ian Maxon, Raj Mohanty, Shengquan Ni, Professor Weifeng Shan, Baihao Wang, Guoxi Wang, Tao Wang, Dr. Zuozhi Wang, Qiancheng Wu, Tejia Zhang, and Tiancheng Zheng. Their support and camaraderie have been invaluable to me.

VITA

Qiushi Bai

EDUCATION

Doctor of Philosophy in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Masters in Computer Science **2023**
University of California, Irvine *Irvine, CA*

Masters in Computer Software and Theory **2012**
Northeastern University *Shenyang, China*

Bachelors in Computer Science and Technology **2010**
Northeastern University *Shenyang, China*

SELECTED PUBLICATIONS

Maliva: Using Machine Learning to Rewrite Visualization Queries Under Time Constraints. **2023**
Qiushi Bai, Sadeem Alsudais, Chen Li, Shuang Zhao. In Proceedings 26th International Conference on Extending Database Technology (EDBT)

Demo of VisBooster: Accelerating Tableau Live Mode Queries Up to 100 Times Faster. **2022**
Qiushi Bai, Sadeem Alsudais, Chen Li. In Proceedings of the Workshops of the EDBT/ICDT Joint Conference

Rainbow: A Rendering-Aware Index for High-Quality Spatial Scatterplots with Result-Size Budgets. **2022**
Qiushi Bai, Sadeem Alsudais, Chen Li, Shuang Zhao. In Proceedings of the 22nd Eurographics Symposium on Parallel Graphics and Visualization (EGPGV@EuroVis)

GSViz: progressive visualization of geospatial influences in social networks **2022**
Sadeem Alsudais, Qiushi Bai, Shuang Zhao, Chen Li. In Proceedings of the 30th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)

Marviq: Quality-Aware Geospatial Visualization of Range-Selection Queries Using Materialization. **2020**
Liming Dong, Qiushi Bai, Taewoo Kim, Taiji Chen, Weidong Liu, Chen Li. In Proceedings of the 2020 International Conference on Management of Data (SIGMOD) Conference

ABSTRACT OF THE DISSERTATION

Improving SQL Performance Using Middleware-Based Query Rewriting

By

Qiushi Bai

Doctor of Philosophy in Computer Science

University of California, Irvine, 2023

Professor Chen Li, Chair

Query performance is critical in database-supported applications where users need answers quickly to make timely decisions. For decades, databases have relied heavily on query rewriting to optimize SQL query performance. However, with the current prevalent use of business intelligence and interactive visualization systems, purely relying on the rewriting capabilities inside databases is insufficient to optimize queries generated by those modern applications. On the one hand, different applications have various performance requirements. Some applications prioritize responsiveness, requiring queries to be executed within strict time constraints, and others may prioritize accuracy. Traditional database-centric approaches fail to exploit such information and adapt to diverse application requirements. On the other hand, developers and domain experts possess valuable insights into the data and query patterns specific to their applications. However, the query optimization techniques customized using domain knowledge can be infeasible inside databases.

In this thesis, we focus on providing middleware-based query-rewriting techniques to help databases seize opportunities to optimize queries.

First, for many applications with stringent response time constraints, such as interactive visualization systems, we propose a machine learning-powered query rewriting framework (called Maliva) to rewrite the queries with various options and help the databases generate efficient plans. Maliva leverages those expensive and high-accuracy query cost estimators to guide their rewriting process. By considering a pre-defined time constraint, Maliva judiciously explores different rewriting options and balances the query planning time and the execution time to find an efficient rewritten query that meets the time constraint.

Second, in many cases, developers want to use their domain knowledge about the applications and datasets to rewrite queries for better performance. We propose a human-centered query rewriting solution (called QueryBooster) to provide users with an express and easy-to-use rule language to define rewriting rules. In addition, QueryBooster allows users to express their rewriting intentions by providing example query pairs. QueryBooster then automatically generalizes them into rewriting rules and suggests high-quality ones to the users.

Finally, to lower the bar of users adopting the proposed rewriting framework, we implemented QueryBooster as middleware-based multi-user system to provide query rewriting between applications and databases as a service. Treating both the applications and databases as black boxes, QueryBooster requires no code modifications to them. To use the service, users only need to replace the database connector between the application and the database with a customized version provided by QueryBooster. The customized connector automatically intercepts application queries and sends them to QueryBooster to rewrite them based on user-defined rewriting rules. QueryBooster's service model brings SQL query rewriting to a new paradigm where (1) users can easily formulate, control, and monitor query rewriting; (2) they can share rewriting knowledge and benefit from the wisdom of the crowd; and (3) they enjoy the non-intrusiveness security and pay-as-you-go convenience.

Chapter 1

Introduction

Data analytics has emerged as an important discipline in today's digital era. As information permeates every aspect of our lives, organizations increasingly recognize the pivotal role of data-driven insights in shaping strategic decisions and driving growth. At the heart of this data revolution lies a crucial element: database-supported applications. Ranging from business intelligence (BI) platforms to visualization systems, these tools enable organizations to uncover meaningful patterns, extract actionable insights, and make informed decisions. As organizations deal with ever-increasing volumes of data, the ability to efficiently retrieve and process data becomes critical. The speed of query execution directly impacts the timeliness of gaining insights, making decisions, and the overall user experience.

For decades, databases have relied heavily on query rewriting to optimize SQL query performance. However, with the current prevalent use of business intelligence and interactive visualization systems, purely relying on the rewriting capabilities inside databases is insufficient to optimize queries generated by those modern applications. There are two main Challenges. **(C1)** On the one hand, different applications have various performance requirements. Some applications prioritize responsiveness, requiring queries to be executed within strict time constraints, and others may prioritize accuracy. Traditional database-centric approaches fail to exploit such information and adapt to diverse application requirements. **(C2)** On the other hand, developers and domain experts possess valuable insights into the data and query patterns specific to their applications. However, the query optimization techniques customized using domain knowledge can be infeasible inside databases.

To seize those opportunities to help databases optimize application queries, we focus on middleware-based query rewriting solutions, which have a few unique advantages. First, unlike the approach of modifying the application or database code, which is time-consuming and error-prone, middleware can be seamlessly integrated into the existing infrastructure. Second, middleware can be tailored to address specific application needs by taking into account factors such as response time and query accuracy to rewrite queries. Finally, sitting in between the application and database, middleware can facilitate domain-specific query optimization techniques by rewriting application queries before sending them to the database with developer-customized rewriting rules. In this thesis, we proposed two solutions to address the aforementioned two challenges.

1. Maliva: Using Machine Learning to Rewrite Visualization Queries Under Time Constraints. Visualization is becoming increasingly important in the Big Data era as a powerful way for people to gain insights from data quickly and intuitively. *Responsiveness* is critical in visualization applications, and a request needs to be served within a time budget, e.g., $500ms$. This requirement is especially challenging when the data volume is large and the user request has ad-hoc conditions on attributes of various types. In Chapter 2, we study the problem of *answering visualization requests with a predetermined time constraint*. We focus on middleware-based solutions, with the advantage that they treat the backend database as a black box without changes and can leverage computing capabilities to do in-situ analytics. We consider rewritings that return exact results and rewritings that return approximate results. We propose a machine learning-powered query rewriting framework (called Maliva) to rewrite the queries with various options and help the databases generate efficient plans. Maliva leverages expensive and high-accuracy query cost estimators to guide their rewriting process. By considering a pre-defined time constraint, Maliva judiciously explores different rewriting options and balances the query planning time and the execution time to find an efficient rewritten query that meets the time constraint. Our experiments on both real and synthetic datasets show that Maliva performs significantly better than a baseline solution that does not do any rewriting in terms of the probability of serving requests interactively and query execution time.

2. QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting. 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. For example, in one of our experiments using Apache Superset to analyze social media tweets on top of a MySQL database, a human-crafted rewriting rule that translates the textual filtering condition from a `LIKE` predicate to a full-text search predicate can improve the query performance by 100 times faster. However, since this rewriting is valid only for a particular dataset, it cannot be adopted by native optimizers inside databases. In addition, existing query rewriting solutions inside and outside databases do not give the users enough freedom to express their rewriting needs. In both Chapter 3 and Chapter 4, we propose a middleware-based human-centered query rewriting system (called QueryBooster) to address this problem.

In Chapter 3, we focus on the core technical aspects of the QueryBooster system. QueryBooster provides users with an express and easy-to-use rule language (VarSQL) to define rewriting rules. In addition, QueryBooster allows users to express their rewriting intentions by providing example query pairs. QueryBooster then automatically generalizes them into rewriting rules and suggests high-quality ones to the users. 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.

In Chapter 4, we focus on the system design aspects of the QueryBooster system to provide query rewriting between applications and databases as a service. QueryBooster requires no code modifications to the applications and databases. To use the service, users only need to replace the connector between the application and the database with a customized connector provided by QueryBooster. This connector automatically intercepts application queries and sends them to QueryBooster to rewrite them based on user-defined rewriting rules. QueryBooster's service model brings SQL query rewriting to a new paradigm where (1) users can easily formulate, control, and monitor query rewriting; (2) they can share rewriting knowledge and benefit from the wisdom of the crowd; and (3) they enjoy the non-intrusiveness security and pay-as-you-go convenience.

In Chapter 5, we conclude the thesis, and we identify several interesting research opportunities as future work.

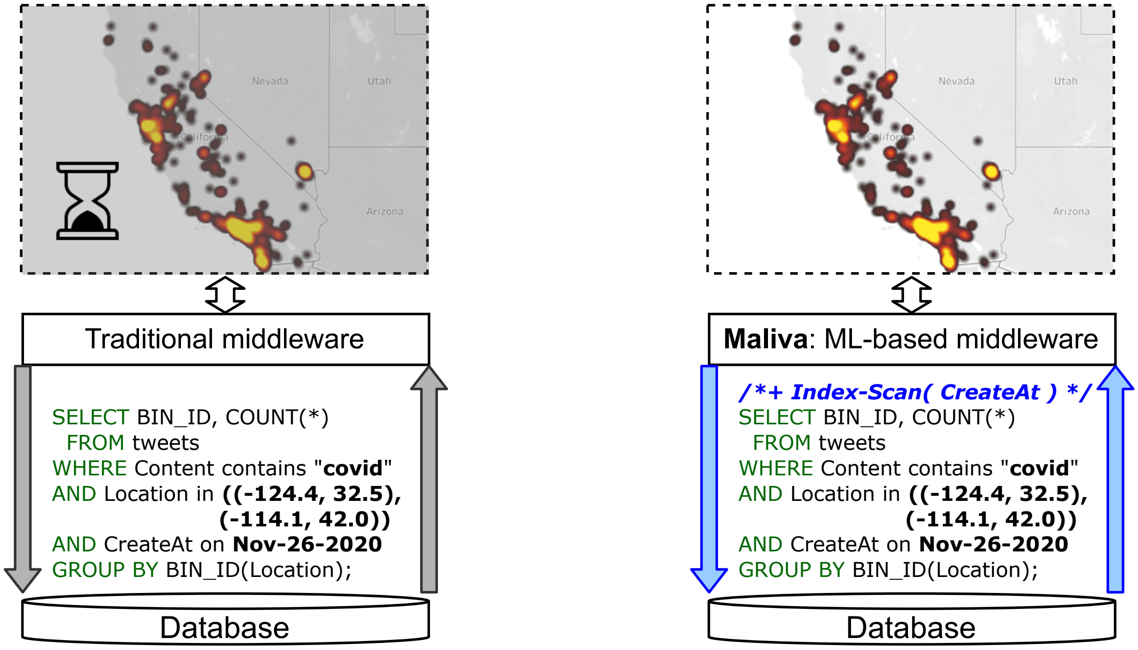
Chapter 2

Using Machine Learning to Rewrite Visualization Queries Under Time Constraints

2.1 Introduction

As a powerful way for people to gain insights from data quickly and intuitively, visualization is becoming increasingly important in the Big Data era. A common architecture to support data visualization has three tiers: a backend database, a middleware layer, and a user-facing frontend. The middleware translates a visualization request to a query (typically in SQL) to the database and sends the query answers to the frontend to display. This architecture is widely used due to its benefits of supporting in-situ analytics at the data source, and utilizing the database's built-in capabilities of efficient storage, indexing, query processing, and optimization. *Responsiveness* is critical in these systems [56, 21, 13], and a request needs to be served within a time budget, e.g., $500ms$. This requirement is especially challenging when the data volume is large, and the user request has ad-hoc conditions on attributes of various types.

In this chapter, we study the problem of *answering visualization requests with a predetermined time constraint*. We focus on middleware-based solutions, with the advantage that they treat the backend database as a black box without changes, and can leverage the computing capabilities to do in-situ analytics. We consider both rewritings that return exact results and rewritings that return approximate results. As a motivating example, consider a system that visualizes social media tweets on the US map with a time constraint of 500ms. Its backend database has a `tweets` table with attributes `Content`, `Location`, and `CreateAt`.



```
SELECT BIN_ID, COUNT(*)
  FROM tweets
WHERE Content contains "covid"
AND Location in ((-124.4, 32.5),
                 (-114.1, 42.0))
AND CreateAt on Nov-26-2020
GROUP BY BIN_ID(Location);
```

```
/*+ Index-Scan( CreateAt ) */
SELECT BIN_ID, COUNT(*)
  FROM tweets
WHERE Content contains "covid"
AND Location in ((-124.4, 32.5),
                 (-114.1, 42.0))
AND CreateAt on Nov-26-2020
GROUP BY BIN_ID(Location);
```

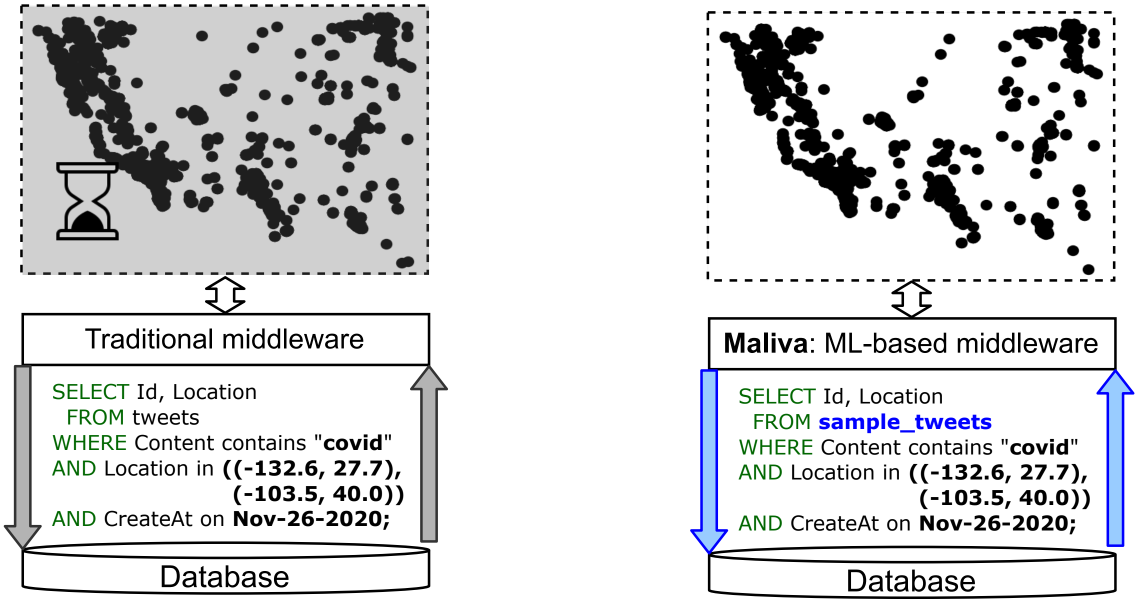
(a) The original SQL query takes 3.35s.

(b) A rewritten query with a hint takes 0.33s.

Figure 2.1: Equivalent rewriting option: adding query hints helps the database compute results within a time budget (500ms).

Equivalent rewriting options. Suppose a user asks for a spatial heatmap of tweets containing the keyword `covid` on the Thanksgiving day of 2020 in a region. The middleware creates a SQL query shown in Figure 2.1(a), which takes 3.35 seconds to execute. For this query, the physical plan generated by the database uses the keyword to access the inverted index on the `Content` attribute to retrieve candidate records, then filters them using the other two conditions. If we rewrite the query to an equivalent query by adding a hint (Figure 2.1(b)), the rewritten query takes only 0.3 seconds, as the hint helps the database generate a more efficient physical plan that uses the temporal filtering condition to access the B+ Tree index on the `CreateAt` attribute.

Approximation rewriting options. Figure 2.2(a) shows another visualization request on a larger region, which takes at least 4.28s for the database to run, no matter what hints we add. In this case, we rewrite the query by using random sampling, resulting in an approximation query that takes only 0.45s to run (see Figure 2.2(b)).



```
SELECT Id, Location
  FROM tweets
WHERE Content contains "covid"
AND Location in ((-132.6, 27.7),
                 (-103.5, 40.0))
AND CreateAt on Nov-26-2020;
```

```
SELECT Id, Location
  FROM sample_tweets
WHERE Content contains "covid"
AND Location in ((-132.6, 27.7),
                 (-103.5, 40.0))
AND CreateAt on Nov-26-2020;
```

(a) The query takes 4.28s (no hints can reduce it).

(b) A rewritten query using a sample table takes 0.45s.

Figure 2.2: Approximation rewriting option: rewriting the query to compute an approximate result within the time constraint.

Why does the database fail? For the query in Figure 2.1(a), there are many reasons the database can fail to generate an efficient plan. One is the estimation error of the query cost due to an underestimation of the keyword `covid`'s selectivity. The cost-estimation problem in optimizers is notoriously hard [58]. For example, in our experiments (Section 3.7), out of the

602 visualization queries that had at least one physical plan that could finish within $500ms$, PostgreSQL failed to choose an efficient plan for 269 queries due to its cost-estimation errors. Although there are many higher-accuracy estimators such as [124, 125, 107, 64, 78, 36, 62], their higher estimation cost prevents them from being adopted by a general-purpose database to meet the visualization need. In particular, for OLTP queries that need to be finished within milliseconds, spending tens of milliseconds for the cost estimation is unacceptable. A key observation is that for visualization applications where requests come with a time constraint, the middleware can afford to spend more time (e.g., 300ms) on the cost-estimation using the high-accuracy estimators to find efficient plans (e.g., within 50ms), while it can still answer requests within a given time constraint (e.g., 500ms).

Challenges. We may enumerate all possible rewritten queries by applying different hints to a given query. We then use one of the aforementioned query-time estimators ("QTE" for short) to estimate the execution time of these rewritten queries and choose the most efficient one. There are several challenges in using this approach in the context of interactive visualization. **(C1)** A main challenge is that the cost of estimating the execution time of a rewritten query can be significant given a tight time constraint. For example, in Bao [62], estimating the execution time of all rewritten queries for one original query can take up to $230ms$ in their experiments. **(C2)** Another challenge is the uncertainty caused by the estimation error of the QTE, and the fact that the backend database may or may not follow the provided hints to generate a physical plan. **(C3)** The third challenge is quality. For queries without equivalent rewritten queries that can meet the time constraint, approximate rewriting options need to be explored. It is critical to maximize the quality of the result while ensuring the query time is within the time constraint.

We address these challenges by introducing a novel machine-learning-based technique called Maliva, which stands for "Machine Learning for Interactive Visualization." The technique formulates the middleware task as a Markov Decision Process (MDP). For a given time budget, we train an MDP agent to balance the planning time and the execution time of the rewritten queries. By learning from previous experiences, the MDP agent judiciously explores different rewriting options, so that the total time (including planning and query execution) is within the time limit. (We address challenge **C1** in Section 2.4.) Using reinforcement learning to train the models, Maliva can handle the uncertainties introduced by the inaccurate time estimation and the fact that the database could ignore the query hints. (We address challenge **C2** in Section 2.5.) By considering visualization qualities of rewritten queries in the reward design of the MDP model, Maliva makes the best effort to maximize the result's quality while ensuring the query time is within the time limit. (We address challenge **C3** in Section 2.6.) Our experiments show that Maliva has a much higher chance (70×) than the original query to generate an execution plan such that the total time is within a time limit. Interestingly, it can also reduce query execution time. Both improvements show the significant benefits of adding learning capabilities to the middleware to support responsive visualization.

The rest of the chapter is organized as follows. After formulating the middleware query-generation problem in Section 2.2, we give an overview of Maliva in Section 2.3. We present the details of this MDP-based solution, including its states, actions, transitions, and rewards (Section 2.4). We present how Maliva trains an MDP agent offline and uses it to generate a rewritten query online (Section 2.5). We generalize the MDP-based solution to be quality-aware by considering approximation rewriting (Section 2.6). Lastly, we report the results of a thorough experimental evaluation of Maliva to show its performance and benefits (Section 3.7).

2.1.1 Related Work

Visualization is a broad topic studied in many communities, and here we focus on efficiency-related works. A survey [31] summarized studies on interactive data analytics and visualization, and there are several recent studies on this topic [44, 87, 53, 50].

Approximate Query Processing (AQP). There are many techniques for computing approximate answers to queries [130, 129, 69, 77, 54, 34, 76, 117, 25, 92, 80, 15]. These approaches focus on developing approximation solutions to compute high-quality visualization. Existing solutions can be adopted as approximation rules in Maliva, such as Sample+Seek [25], which generates error-bounded visualization results by running queries on a small sample table.

Datacube-based approaches. Related studies include [55, 24, 120, 57, 43, 21, 46, 68]. In these approaches, the predefined cube intervals cannot support visualization queries with arbitrary numerical range conditions. The proposed Maliva system efficiently computes results for visualization queries with arbitrary conjunctive selection conditions.

Progressive visualization. There are solutions to show visualization results progressively [67, 42, 20, 29, 21, 15]. For instance, DICE [21] uses random and stratified samples to present an approximate result and then incrementally updates the result. These progressive visualization systems can adopt the proposed Maliva middleware to further optimize the intermediate queries to increase their efficiency.

Prefetching-based approaches. Techniques including [12, 126, 97, 3] accelerate visualization queries by prefetching or caching their results. For example, ForeCache [12] divides visualizations into tiles and prefetches them based on predicted user behaviors. Maliva is orthogonal to these techniques, and it can be adopted by them to further optimize the database queries.

Visualization using big data systems. These techniques use Hadoop, Spark, and Hive to support visualizations [127, 15, 110, 26, 16]. For instance, HadoopViz [26] and GeoSparkViz [127] use Hadoop and Spark to generate high-resolution visualizations. Their focus is on offline construction, not on an interactive visualization for queries with ad-hoc conditions. The proposed Maliva middleware technique is complementary to these solutions.

ML for visualization. A survey by Wang et al. [118] summarized studies of applying ML techniques to different stages during the whole visualization pipeline. Examples are [59, 119] for data cleaning and preparation and [40, 89, 60] for visualization recommendation. Our proposed system focuses on applying ML techniques to solve performance issues at the middleware.

ML-based query optimization. ML has recently used in database optimizers [62, 63, 128, 64, 101, 51, 115], selectivity estimation [78, 36], and cost estimation [107]. **Comparison with Bao**: The recent Bao technique [62] uses hints to generate optimized queries by modeling the optimization as a multi-armed bandit problem. It applies Thompson sampling to minimize the training time and maximize the accuracy of its neural-network-based query time estimator (QTE). We have a detailed discussion about the differences between Maliva and Bao in Section 2.6.3. We also conducted extensive experiments in Section 3.7 to compare their performance, and the results show Maliva outperformed Bao in various metrics (See Section 2.7.6).

2.2 Problem Formulation

Visualization architecture. We consider a typical three-tier data-visualization system that consists of a backend database, a middleware layer, and a frontend. For each frontend visualization request, let Q be the *original* SQL query for the request. Let τ be a time limit that quantifies the expected responsiveness of the system. Ideally, we want the total delay, from the time the user submits a request to the time the result is shown on the frontend, to be within τ. The original query Q may not meet the time-limit constraint when the backend database cannot generate a physical plan that is fast enough. To solve this problem, Maliva rewrites Q with two kinds of options: *query hints* and *approximation rules.* By adding a query hint to Q, Maliva can help the backend database generate an efficient physical plan that computes the result within the time limit. For expensive queries where no physical plan can meet the time limit, Maliva can add an approximation rule to the original query such that the backend database computes an approximate result to trade the visualization quality for responsiveness. Note that the proposed approach also works in a more general setting of approximate query processing (AQP) where a time constraint is given.

Query hints. A *query hint* in a database is an addition to the SQL statement that instructs the database engine on how to execute the query. For example, a hint may tell the engine to use or not to use an index (even if the query optimizer would decide otherwise) [38]. A query hint does not change the semantic meaning of the query, i.e., the result computed by the database engine with the hint remains the same. Databases such as AsterixDB [9], MySQL [71], Oracle [75], PostgreSQL [86], and SQL Server [105] support a variety of query hints. For example, in Figure 2.3(b), Maliva adds two hints *+ Index-scan(t CreateAt)* and *Nest-Loop-Join(t u)* to the original query. They suggest the engine to use the index on the `CreateAt` attribute to scan the table `t`, and do a nest-loop join on tables `t` and `u`.

(a) Original Query (Q)

```
SELECT BIN_ID, COUNT(*)
  FROM tweets t, users u
WHERE t.Content contains "covid"
    AND t.Location in ((-124.4, 32.5),
                       (-114.1, 42.0))
    AND t.CreateAt on 'Nov-26-2020'
    AND u.TweetCnt in [100, 5000]
    AND t.user_id = u.id
  GROUP BY BIN_ID(t.Location);
```

(b) Rewritten Query (RQ)

```
/*+ Index−scan(t CreateAt),
    Nest−Loop−Join(t u) */
SELECT BIN_ID, COUNT(*)
  FROM tweetsSample20 t, users u
WHERE t.Content contains "covid"
    AND t.Location in ((-124.4, 32.5),
                       (-114.1, 42.0))
    AND t.CreateAt on 'Nov-26-2020'
    AND u.TweetCnt in [100, 5000]
    AND t.user_id = u.id
  GROUP BY BIN_ID(t.Location);
```

Figure 2.3: A original query and a rewritten query.

Approximation rules. An *approximation rule* is a method to rewrite the original SQL query to compute an approximate result, and the new query takes less time. There are various approximation rules available in database systems, such as adding a *"Limit"* clause, applying a SQL-standard *"TableSample"* operator on a table, or substituting a table with a smaller table randomly sampled from the original table. For example, in Figure 2.3(b), Maliva rewrites the original query by substituting the table `tweets` with a sample table `tweetsSample20` with 20% randomly selected records.

Now we formally define rewriting options, rewritten queries, and the query-rewriting problem.

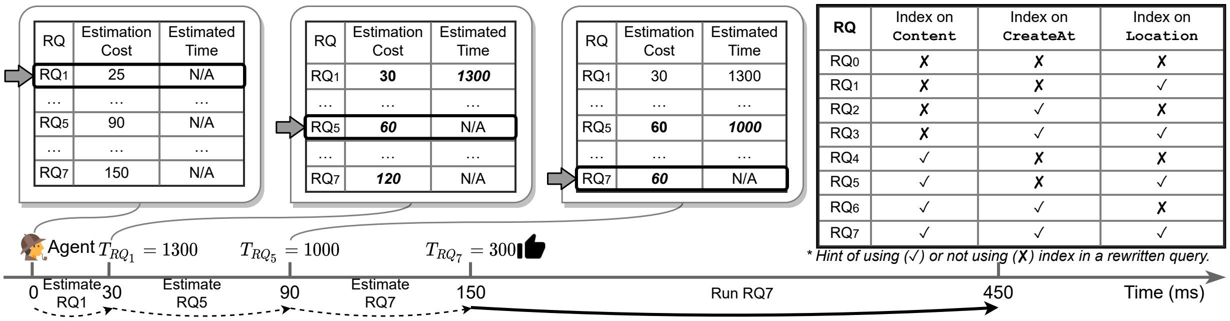
Definition 2.1. *(Rewriting Option) Let H be a set of query-hint sets and A be a set of approximation-rule sets. A rewriting option ("RO" for short) is a tuple (h, a), where $h \in H$ and $a \in A$. Note that both h and a can be the empty set $\emptyset$.*

For instance, the rewriting option in Figure 2.3(b) is a tuple with a query-hint set of "use the index on `CreateAt` and do a nest-loop join on `t` and `u`" and an approximation-rule set of "substituting the table `tweets` with the sample table `tweetsSample20`". We assume the user-defined candidate set of rewriting options does not contain invalid query-hint sets or approximation rules. A query-hint set is considered to be invalid if it contains conflicting hints, e.g., *Nest-Loop-Join(t u)* and *Hash-Join(t u)*.

Definition 2.2. *(Rewritten Query) Given an original SQL query Q and a rewriting option RO, a rewritten query ("RQ" for short) is a new SQL query generated by applying RO onto Q. If RO $= (\emptyset, \emptyset)$, then RQ $= Q$.*

For example, Figure 2.3(b) is a rewritten query for the original query in Figure 2.3(a).

Query-rewriting problem. Given a visualization request's original SQL query Q, and a time limit τ, we want to generate a rewriting option, such that the total time of the corresponding rewritten RQ, including planning and query execution, is within τ and the quality of RQ's result is maximized. To quantify the quality, we assume a given visualization quality function F. Let $r(Q)$ be the result of the original query Q and $r(RQ)$ be the result of the rewritten query RQ. Then $F(r(Q), r(RQ))$ computes the quality of $r(RQ)$.



RQ	Estimation Cost	Estimated Time
RQ1	25	N/A
…	…	…
RQ5	90	N/A
…	…	…
RQ7	150	N/A

RQ	Estimation Cost	Estimated Time
RQ1	**30**	***1300***
…	…	…
RQ5	***60***	N/A
…	…	…
RQ7	***120***	N/A

RQ	Estimation Cost	Estimated Time
RQ1	30	1300
…	…	…
RQ5	**60**	***1000***
…	…	…
RQ7	***60***	N/A

RQ	Index on `Content`	Index on `CreateAt`	Index on `Location`
RQ0	✗	✗	✗
RQ1	✗	✗	✓
RQ2	✗	✓	✗
RQ3	✗	✓	✓
RQ4	✓	✗	✗
RQ5	✓	✗	✓
RQ6	✓	✓	✗
RQ7	✓	✓	✓

** Hint of using (✓) or not using (✗) index in a rewritten query.*

Figure 2.4: The Query Rewriter acts like an agent who makes a sequence of decisions to generate a rewritten query (with a total time $\leq 500ms$). At time 0, the agent considers rewritten query RQ_1 due to its low estimation cost (estimated $25ms$, the actual $30ms$ is updated once RQ_1 is explored). After estimating its execution time ($1,300ms$), the agent knows that RQ_1 is not viable since the total time is longer than $500ms$. The estimation of RQ_1 affects the costs for estimating RQ_5 and RQ_7. The agent explores RQ_5 and then RQ_7. With the estimated execution time being $300ms$ and the elapsed time being $150ms$, RQ_7 is decided as a viable rewritten query because the total time ($450ms$) is within $500ms$.

In Sections 2.3, 2.4, and 2.5, we study the case of using query hints only (i.e., without changing query results). In Section 2.6, we study the case where approximation rules are also used.

2.3 Maliva: ML-based Query Rewriting

We now introduce the middleware technique called "Maliva" to solve the aforementioned query-rewriting problem. We first give an overview of the technique, then use an example to explain the details.

Overview. As illustrated in Figure 2.5, Maliva rewrites the original SQL query to answer a visualization request within a time budget. It considers a predefined set of rewriting options, which we denote as $\Omega = \{RO_1, \ldots\}$. For each RO_i, the rewritten query is denoted as RQ_i. The set of candidate rewritten queries is $\Phi = \{RQ_1, \ldots\}$.

Maliva has a *Query Rewriter* that enumerates possible RQs and uses a *Query Time Estimator*

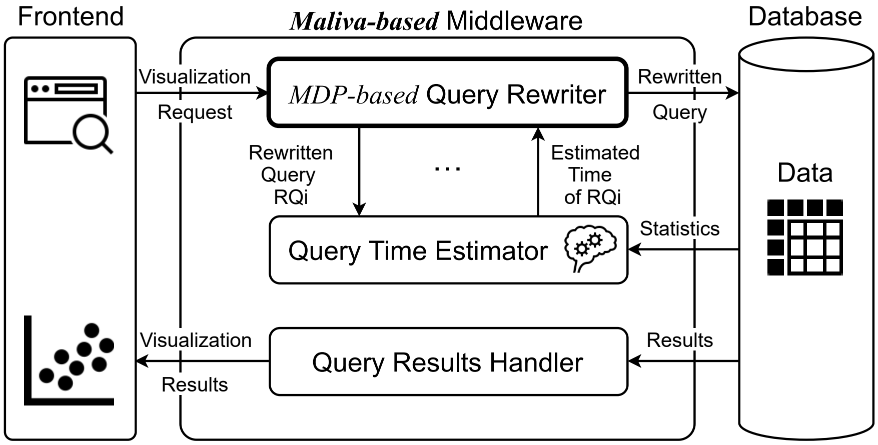


Figure 2.5: Overview of Maliva.

(QTE) to estimate the execution time of each of them. The *Query Rewriter* uses the best effort to choose an RQ such that the total time, including the planning process and query execution, is within the time budget τ. Such an RQ is called *viable.* The middleware then sends the rewritten query to the database. The *Query Result Handler* sends the retrieved result to the frontend to visualize.

Naïvely enumerating all available RQs in Φ is computationally prohibitive due to two reasons. First, the cost of *Query Time Estimator* to estimate the execution time of a rewritten query is not negligible. For instance, in some cases it could take up to $70ms$ [107] on a $7GB$ dataset or $300ms$ [124] on a $10GB$ dataset. Second, the number of RQs increases exponentially when the number of applicable indexing choices increases. For example, consider a selection query on a table with filtering conditions on m attributes, and the database has an index on each attribute. The number of query-hint sets in H would be 2^m, since the database can use any subset of the m indexes to do filtering and then intersect the record lists to compute the final result. Therefore, the *Query Rewriter* needs to balance the exploration time for query estimation and the execution time of each chosen RQ to find a viable RQ.

An example. Maliva views query rewriting as a Markov decision process (MDP) [108] and adopts machine learning (ML) to solve this problem. We use the running example in

Section 2.1 to illustrate how Maliva uses an MDP agent to make a sequence of decisions to find a viable RQ. For simplicity, we assume the rewrite-options (RO) set to involve query hints only. We will generalize the technique to consider approximation rules in Section 2.6. As shown in Figure 2.4, a query has three selection conditions on three attributes, and each of which has an index. Suppose in the set H of query-hint sets, each attribute has a query hint of using or not using the index. Thus, we have $2^3 = 8$ query-hint sets to choose from. The agent makes a sequence of decisions to estimate the execution times of several rewritten queries and find a viable one RQ_7 (that uses the indexes on all three attributes). Next, we present the details of this MDP-based technique.

2.4 MDP Model for Adding Query Hints

In this section, we present the details of using an MDP model in Maliva to solve the query-rewriting problem and discuss how to implement the Query Time Estimator (QTE).

2.4.1 MDP Model for Query Rewriting

MDP [108] is a formalization of sequential decision-making problems where an agent learns to achieve a goal from interaction with an environment. At each time step, the agent is in a state s, and chooses an action a available in state s. The environment transits the agent to a new state s', and gives the agent a corresponding reward $R(s, a)$. To train an MDP agent is to find a good policy π_* such that if the agent follows the policy to choose an action for each state, it maximizes the total reward.

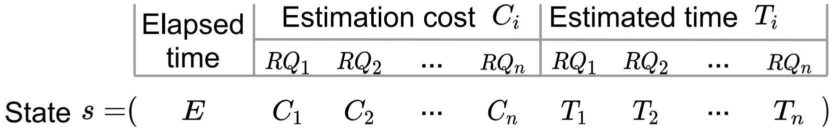
Motivation of using MDP. In the example shown in Figure 2.4, we observed some unique properties in the decision-making process. (1) The rewritten queries considered earlier can influence the agent's later decisions because exploring earlier queries also provides additional information. (2) Since the agent's goal is to find a viable rewritten query of which the total time is within the time budget, it has to balance between the time spent to estimate rewritten queries and their execution time. Based on these observations, instead of machine learning models that decide on a rewritten query directly by looking at the query itself, MDP is a promising candidate mathematical model that captures those properties.

We use the MDP model to solve the query-rewriting problem. For simplicity, we first focus on the case where rewriting options do not contain any approximation rules, which means no rewritten queries have quality loss. We will generalize the technique to consider approximation rules in Section 2.6. Without considering quality loss, the MDP agent learns to maximize the chance of finding a viable rewritten query for a given visualization request. The agent takes a sequence of actions, and each action chooses an RQ to explore. That is, it asks the query time estimator (QTE) to estimate the execution time of the RQ. The agent chooses an RQ based on the current state, and considers the future cost it needs to pay and the execution time of RQs already explored. The agent gets a bonus if it finds a viable RQ, or a penalty if it runs out of time. In the offline phase, by analyzing queries in the training workload, the agent learns to maximize the chance to receive a bonus. In the online processing phase, given a new query, the agent decides which RQ to explore in each step to receive a bonus in the end. Now we present the details of how to use MDP to model the process of choosing RQs.

States. A state represents the past decisions, based on which the agent decides an RQ to consider next. Suppose we are given a predefined set of n ROs, i.e., $\Omega = \{RO_1, \ldots, RO_n\}$. Correspondingly, we have n candidate RQs, denoted as $\Phi = \{RQ_1, \ldots, RQ_n\}$. A state is a vector

$$s = (E, C_1, C_2, \ldots, C_n, T_1, T_2, \ldots, T_n),$$

which includes three pieces of information, as shown in Figure 2.6. (1) The elapsed time (E) captures how much time we have spent. (2) The estimation cost (C_i) for each possible rewritten query RQ_i captures how much time is needed for the agent to estimate its running time. Each C_i is initialized with a rough estimation collected offline and updated during the online planning phase. Note that the MDP state does not require the initial C_i values to be accurate, and a rough estimation from history statistics suffices. The actual estimation costs will be collected while the MDP agent processes a query, as will be described soon in the definition of *Transitions*. (3) T_i is the estimated time for each already explored RQ_i. Each T_i is initialized with a 0 value until it is filled with an estimated execution time.



	Elapsed time	Estimation cost C_i				Estimated time T_i			
		RQ_1	RQ_2	...	RQ_n	RQ_1	RQ_2	...	RQ_n
State $s =($	E	C_1	C_2	...	C_n	T_1	T_2	...	T_n)

Figure 2.6: An MDP state in Maliva.

We assume for each rewritten query RQ_i, collecting the physical plan and its statistics (e.g., cardinality and cost estimations of each operator) is done by the QTE, and its time is captured by the MDP model's estimation cost (C_i). We assume the rewritten queries' physical plans and statistics are not available to the MDP model. Thus, the proposed MDP model is general, and can be applied to any query template (a query template has a fixed set of join tables, a fixed set of filtering attributes, and a fixed set of projection attributes) with any predefined query-hint set. A natural question is that without the statistics of the explored RQs stored in the state, how can the MDP model make a good decision on which RQ to choose next? Our answer is that the execution time of a rewritten query implicitly captures the statistics of the physical operators (e.g., the cost of doing an R-Tree index scan on the `Location` attribute). By keeping the estimated execution time of each explored RQ in its state, the MDP model can learn the correlations of the execution times between different rewritten queries and make good decisions. Note these correlations are dependent on the dataset (e.g., data distribution, available indexes, etc.) and the database configuration (e.g., RAM budget). Thus the model needs to be re-trained if the dataset or the database configuration changes.

Actions. An action, denoted as a, is to explore an RQ next. For each RQ, the agent asks the `QTE` to estimate its execution time. Meanwhile, the agent needs to pay a cost as it takes time for the QTE to extract query features, possibly by collecting online statistics from the database, and running the estimation model to do the estimation. In the running example, at time 0, the agent decides to explore RQ_1. It asks the `QTE` to estimate RQ_1's execution time.

Transitions. A transition function defines how the environment computes the next state, given the agent's action in the current state. Let the RQ considered by action a in state s be RQ_i. We define the transition function $\mathcal{T}$ as follows. First, the `QTE` (a black-box to the MDP agent) estimates the time of RQ_i, and we add the estimated time T_i to the state. Second, the estimation costs for other RQs could change. In the running example, to estimate RQ_1 that uses the R-Tree index on the `Location` attribute, we need to collect the spatial filtering condition's selectivity on the `Location` attribute. To estimate RQ_5 that uses both the inverted index on the `Content` attribute and the R-Tree index on the Location attribute, we need to collect the selectivity values of the filtering conditions on both attributes. After the agent takes the RQ_1 action, we update the estimation cost of RQ_1 to be the actual time it costs and update the approximated estimation cost of RQ_5 by excluding the cost to collect the selectivity value of the spatial filtering condition. Figure 2.7 shows an example of state transition. After estimating the time of RQ_1, we add the estimated time $1,300ms$ of RQ_1 to the state, update the estimation cost for RQ_1 from the estimated $25ms$ to the actual $30ms$, and update the estimation cost for RQ_5 from the previous estimated $90ms$ to the new estimated $60ms$. Lastly, the estimation takes time $\hat{C_i}$, and we add it to the elapsed time so far to indicate how much time the agent has spent exploring different RQs. Note that the $\hat{C_i}$ is the actual cost of estimating RQ_i, which could be different from C_i because C_i is an approximated cost for estimating RQ_i. For example, if the QTE estimates the cost of RQ_1 by running a query on a sample table, the actual cost of QTE estimating RQ_1 is only obtained after we pay the cost.

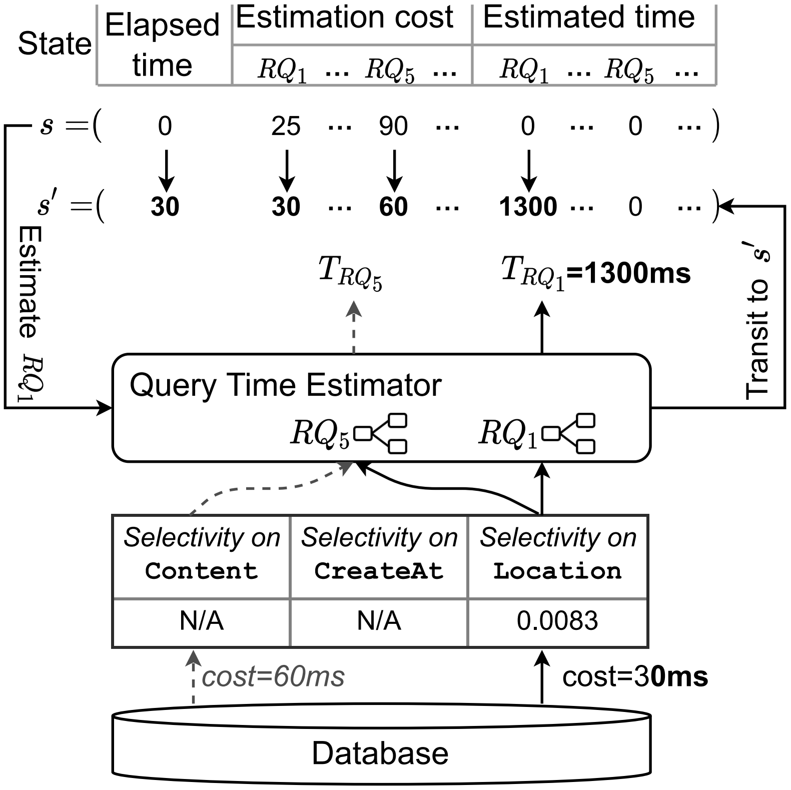


Figure 2.7: Transition after estimating execution time of RQ_1.

Rewards. A reward function defines the agent's immediate gain when it takes a particular action a in a given state s. In our setting, consider two cases to compute the reward function. (1) The first case is when the agent is at an intermediate state where it still has time for planning but has not yet found a viable rewritten query. In this case, the agent should not be awarded or punished since it has not made a decision yet. Thus the reward value is 0. (2) The second case is when the agent is at a termination state where it decides the rewritten query $\hat{RQ}$, runs it against the database, and collects the execution time $\hat{T}$ of $\hat{RQ}$. In this case, the agent should be awarded if the total time (including both the planning time and rewritten query execution time) is less than the time budget, or punished if the total time is more than the budget.

The agent decides on a rewritten query by considering three situations. The first one is that the agent finds an RQ to be viable based on the estimation of the QTE before running out of time. For example, in Figure 2.4, after spending $150ms$ for planning, the agent decides RQ_7 as the chosen rewritten query, since the predicted total time of RQ_7 is $450ms$, which is within the $500ms$ budget. The second situation is when the agent uses up the time budget and has to stop planning. The third situation is when the agent has exhausted all candidate rewritten queries and has to decide which RQ to choose. In the latter two situations, the agent chooses the fastest RQ known so far as the final decision.

Formally, suppose the generated rewritten query by the agent is $\hat{RQ}$ and the actual running time of query $\hat{RQ}$ is $\hat{T}$. Then the reward function $\mathcal{R}(s, a)$ is defined as follows,

$$\mathcal{R}(s, a) = \frac{(\tau - s.E - \hat{T})}{\tau}, \tag{2.1}$$

where $s.E$ denotes the elapsed time so far in state s. If the total time $s.E + \hat{T}$ is less than the time budget τ, which makes $\mathcal{R}(s, a)$ positive, then the agent receives a reward. The faster the rewritten query is, the larger the reward will be. On the other hand, if the total time exceeds the time budget, which makes $\mathcal{R}(s, a)$ negative, then the agent receives a penalty. The slower the rewritten query is, the larger the penalty will be. Thus, guided by the reward function, the MDP model will learn to find an efficient rewritten query as soon as possible.

2.4.2 Query Time Estimator (QTE)

Take the sampling-based QTE described in [124] as an example. It first builds an analytical cost model (e.g., linear regression model), and uses it to estimate the execution time of a rewritten query by collecting its statistics online. Specifically, it estimates the selectivity values of the query conditions by running `count(*)` queries on a small sample table, provides the values as input features to the cost model, and uses the model's prediction as the query's execution-time estimation. There are also other possible solutions in the literature [125, 107, 64] that can be used by Maliva. Note that QTEs are the focus of this chapter, and Maliva leverages a given QTE intelligently to balance the planning time and the query execution time.

2.5 Training and Using the MDP Agent

In this section, we discuss how to train the MDP agent offline in Maliva on a workload of visualization requests and use it to generate a viable rewritten query online.

2.5.1 Training the MDP Agent

Suppose we have a workload of queries $W = [q_1, q_2, \ldots, q_m]$. Our goal is to find an optimal policy π^* such that for any query $q_i \in W$, the agent following policy π^* maximizes the chance to generate a viable rewritten query. We adopt the *deep Q-learning* algorithm [65] for finding an optimal policy for the MDP agent. Its main idea is to use a neural network (called *Q-network*) to represent a policy π. Given an input of a state vector, the q-network outputs a *Q-value* [122] for each possible RQ in the state. A higher q-value means that the rewritten query is more likely to be viable given the current information. Its training process includes two main steps. The first step is to generate a set of experiences by exploring different planning sequences for queries in the workload repeatedly. The second is to replay those experiences to update the q-network's weights gradually such that the q-network can approximate the q-values of the optimal policy for each state-action pair.

Training an MDP agent for query rewriting. Algorithm 1 details how we train the MDP agent. To apply deep q-learning, we generate the replay memory M of experiences. For a given visualization query workload $W = [q_1, q_2, \ldots, q_m]$, we generate a set of experiences. Each experience is a 4-tuple

$$(s, a, s', r'),$$

where the agent in a state s estimates the time of the hinted query represented by an action a and observes the next state s' with a reward r'. Note that different queries can have different optimal policies. Our goal is to learn an optimal policy for the whole workload. We let the agent explore all the queries in the workload W in multiple iterations until the policy converges or the number of runs exceeds a maximum threshold. In each iteration, we shuffle the order of queries to reduce the bias caused by earlier queries on the exploration direction of later queries. For each query q in W, we let the agent complete a sequence of decisions. At each step, it selects an RQ to estimate. It pays the cost to estimate the rewritten query's execution time, transits to the next state, and receives an immediate reward. The agent repeats the process until it reaches a termination state (line 9) in one of the three cases. The first case is when the estimated time $T(a)$ of the rewritten query in action a suggests it is potentially viable, i.e., $s.E + T(a) \leq \tau$. The second case is when the agent runs out of time,

Algorithm 1: Training an MDP agent

```
Input: A query workload $W = [q_1, q_2, \ldots, q_m]$
       A transition function $\mathcal{T}$
       A reward function $\mathcal{R}$
       A time budget $\tau$
Output: An agent's policy $\pi$
Replay memory $M \leftarrow \{\}$ with capacity $\mathcal{C}$;
Initialize policy $\pi$ with random weights;
while $\pi$ does not converge do
    $W \leftarrow$ shuffle the queries in $W$;
    for each query $q$ in $W$ do
        State $s \leftarrow (0, C_1, C_2, \ldots, C_n, 0, 0, \ldots, 0)$;
        Remaining set $\rho \leftarrow$ query $q$'s all possible RQs $\{RQ_1, RQ_2, \ldots, RQ_n\}$;
        Reward $r \leftarrow 0$;
        while ($s$, $\tau$, $\rho$) is not at a termination state do
            $f \leftarrow$ generate a random number from [0,1];
            if $f < \epsilon$ then
                $a \leftarrow$ a random RQ from $\rho$;
            else
                $a \leftarrow \arg\max_{RQ_i \in \rho} \mathcal{Q}^{\pi}(s, RQ_i)$;
            end
            // Estimate query $a$ and transit to state $s'$
            $s' \leftarrow \mathcal{T}(s, a)$;
            // Compute the immediate reward
            $r' \leftarrow \mathcal{R}(s, a)$;
            Store experience tuple $(s, a, s', r')$ in $M$;
            // Remove query $a$ from the remaining set $\rho$
            $\rho \leftarrow \rho - \{a\}$; $s \leftarrow s'$; $r \leftarrow r'$;
        end
        Update $\pi$ using a random sample from $M$;
    end
end
```

i.e, $s.E \geq \tau$. The third case is when the agent has exhausted all possible RQs, i.e., $\rho = \emptyset$.

When the agent decides which RQ to explore at each step (lines 12 to 14), we adopt an ϵ-greedy strategy [65, 112] to balance between the exploration of RQs with uncertain values and the exploitation of RQs known with high values. With an ϵ probability, the agent chooses a random RQ that has not been considered before (line 12). Otherwise, it selects an RQ that has not been explored with the highest q-value based on the current policy weights (line 14). We start with a high probability (ϵ) of exploration and gradually decrease it to favor exploitation with the training progress.

Once an RQ is decided by the agent as an action a, we call the transition function $\mathcal{T}$ (Section 2.4.1) to transit the agent to the next state s' (line 16). We estimate the query a's running time and update the new state s' by adding the estimated time for a, adding the cost to the elapsed time, and modifying the costs of affected RQs. We then call the reward function $\mathcal{R}$ (Section 2.4.1) to compute an immediate reward r' for estimating the RQ in a (line 17). To this end, we have generated a new experience tuple (s, a, s', r'), and store it in the replay memory M (line 18). When M reaches its capacity $\mathcal{C}$, we replace existing experiences in a FIFO manner.

After processing a query, we update the policy π following the original deep q-learning algorithm [65] (line 21). We sample a random subset M' of experiences from M. For each experience tuple (s, a, s', r') in M', we first compute the target q-value y of the state-action pair (s, a) using the Bellman equation [122]. We then update the weights in policy π by minimizing the loss value L between the target q-value y and the current q-value, where L is defined as:

$$L = (\mathcal{Q}^{\pi}(s, a) - y)^2.$$

We keep updating the policy π until it converges, i.e., the total accumulated reward of the training workload w does not improve much in new iterations (e.g., less than 1%).

Accommodating estimation inaccuracy using MDP. One advantage of using the MDP framework where an approximate QTE may give inaccurate estimations is its tolerance of the inaccuracy. The MDP model captures the uncertainty in two places. One is the transitions between states that store the estimated times of explored RQs. Although estimated times can have errors, statistically, after learning from the historical queries, the agent understands which action has the highest expected total reward. Another place is the reward definition, where the penalty for making a wrong decision will lead the agent to understand the QTE's mistakes and avoid them in the future.

2.5.2 Using MDP to Rewrite Queries Online

After we train an MDP agent, the query rewriter utilizes the agent to generate a rewritten query for a new visualization query q online. Algorithm 2 shows the pseudo-code. Starting from an initial state s, we use the trained policy π to compute the q-values for all the RQs and select the one with the highest q-value as the action a (line 5). We then estimate the running time of query a and transit to state s' (line 6). We compute the immediate reward r' for estimating RQ in a (line 7). If the action a is a potentially-viable RQ (line 9), we output the query $\hat{RQ_i}$ in a as the generated rewritten query. Otherwise, we run out of time for the remaining RQs (line 11). Then we select the rewritten query RQ_j with the minimum execution time estimated so far and output it. If neither cases happen, we repeat the above process.

2.6 Approximation Rewriting Options

In this section, we generalize Maliva by considering rewriting options with approximation rules. Recall that using a query-hint set to rewrite an original query Q into an RQ can help

Algorithm 2: Generating an RQ online

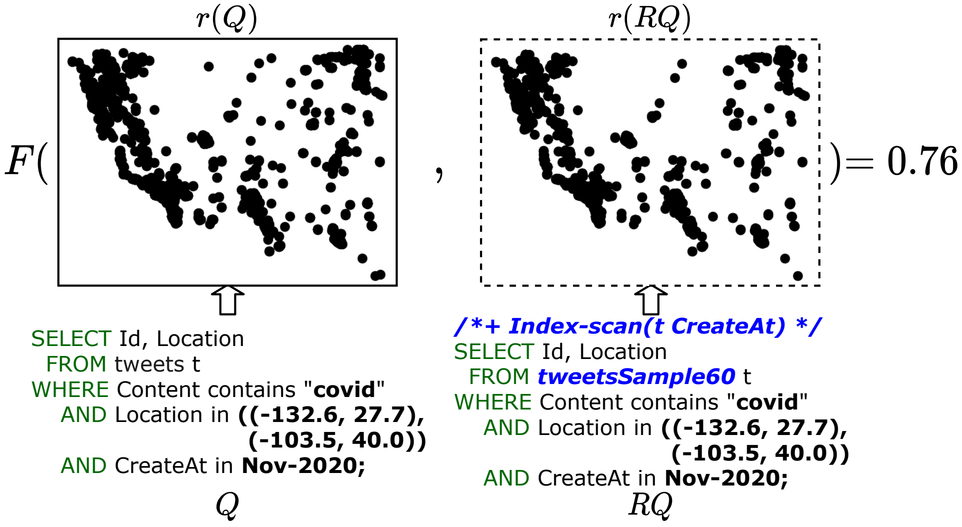
```
Input: A new query $q$
       A trained policy $\pi$
       A transition function $\mathcal{T}$
       A reward function $\mathcal{R}$
       A time budget $\tau$
Output: An RQ
State $s \leftarrow (0, C_1, C_2, \ldots, C_n, 0, 0, \ldots, 0)$;
Remaining set $\rho \leftarrow$ query $q$'s all possible RQs $\{RQ_1, RQ_2, \ldots, RQ_n\}$ ;
Reward $r \leftarrow 0$;
while $True$ do
    // Select a query with the highest q-value predicted by $\pi$
    $a \leftarrow \arg\max_{RQ_i \in \rho} \mathcal{Q}^{\pi}(s, RQ_i)$;
    // Estimate query $a$ and transit to state $s'$
    $s' \leftarrow \mathcal{T}(s, a)$;
    // Compute the immediate reward
    $r' \leftarrow \mathcal{R}(s, a)$;
    // Remove query $a$ from the remaining set $\rho$
    $\rho \leftarrow \rho - \{a\}; \leftarrow s'; r \leftarrow r'$;
    if $s.E + T(a) \leq \tau$ then
        return $\hat{RQ}_i$ represented by $a$;
    if $s.E \geq \tau$ or $\rho = \emptyset$ then
        return $RQ_j$ with the minimum execution time estimated in $s$;
end
```

the database generate an efficient physical plan that computes the actual result without any approximation. However, for expensive queries where no physical plan can meet the time constraint, by applying an approximation-rule set to Q, Maliva can generate an RQ that efficiently computes an approximate result within the time budget. We first extend the MDP model in Section 2.4 to consider approximation rules. We then discuss two approaches to applying the MDP model to implement a quality-aware query rewriter. The quality-aware query rewriter makes the best effort to generate a viable rewritten query and maximize the result's quality. In the end, we discuss the trade-offs between the two approaches.

2.6.1 Quality-Aware MDP Model

Consider the case where the rewriting options contain both query hints and approximation rules. A rewritten query can return an approximate result with quality loss. We need to let the MDP agent learn to maximize the chance to generate a viable rewritten query and maximize the quality of the query result simultaneously. To quantify the quality of a rewritten query, we assume a given visualization quality function F. Let $r(Q)$ be the result of the original query Q, and $r(RQ)$ be the result of the rewritten query RQ. Then $F(r(Q), r(RQ))$ computes the quality of $r(RQ)$. For example, suppose we use the Jaccard similarity function to measure the quality of an approximate result. Figure 2.8 shows that the quality of the scatterplot visualization result of an approximate rewritten query RQ compared to the original query Q is 0.76. Note that Maliva does not have restrictions on quality functions, and many functions can be used, such as VAS in [76] for scatterplots and the function of distribution precision in [25] for pie charts.

Reward function for a quality-aware MDP model. To achieve the goal of guiding the MDP agent to learn to maximize the chance to generate a viable rewritten query and maximize the quality of the query result simultaneously, we extend the definition of the reward



```
SELECT Id, Location
  FROM tweets t
WHERE Content contains "covid"
    AND Location in ((-132.6, 27.7),
                     (-103.5, 40.0))
    AND CreateAt in Nov-2020;
```

```
/*+ Index-scan(t CreateAt) */
SELECT Id, Location
  FROM tweetsSample60 t
WHERE Content contains "covid"
    AND Location in ((-132.6, 27.7),
                     (-103.5, 40.0))
    AND CreateAt in Nov-2020;
```

Figure 2.8: The quality of RQ compared to Q using a Jaccard-based quality function as an example.

function in Section 2.4. Recall that the learning goal of an MDP agent is to maximize the accumulative reward. In Section 2.4, once the agent decides on a rewritten query, it receives a reward that reflects the query performance in terms of the total running time. Guided by the reward, the agent learns to generate a viable rewritten query quickly. Similarly, the MDP agent can also learn to quickly generate a viable rewritten query with a high result quality if the final reward reflects both the decided rewritten query's efficiency and quality. The main idea is to combine the efficiency defined in Section 2.4 and the quality. The new reward function is a weighted summation of both. Formally, suppose the generated rewritten query by the agent is $\hat{RQ}$ and the actual running time of query $\hat{RQ}$ is $\hat{T}$. Then the new reward function $\mathcal{R}(s, a)$ is defined as follows:

$$\mathcal{R}(s,a) = \beta\frac{(\tau - s.E - \hat{T})}{\tau} + (1-\beta)F\big(r(Q), r(\hat{RQ})\big). \tag{2.2}$$

The term $\frac{(\tau - s.E - \hat{T})}{\tau}$ represents the efficiency of the rewritten query in terms of running time compared to the time budget. The function $F(r(Q), r(\hat{RQ}))$ represents the quality of the

RQ's result. Note that computing F could be expensive since the actual result $r(Q)$ of the original query is required. However, we only need to pay the cost in the offline training phase once. In the online phase, we don't need to compute the F value for a new query when we use the MDP model to explore different RQs. Since the MDP model learns from the final reward values only, we do not require every query to use the same quality function. In particular, different quality functions can be applied for different training queries to evaluate their visualization qualities, e.g., some queries are visualized as scatterplots and others as heat-maps. $\beta \in [0, 1]$ is a parameter that indicates how important the running time is compared to the result quality.

2.6.2 Quality-Aware Query Rewriter

Now we discuss how to apply the extended MDP model to implement a quality-aware query rewriter. We present the technical details of two approaches and discuss their pros and cons. We will show the evaluation results in Section 3.7.

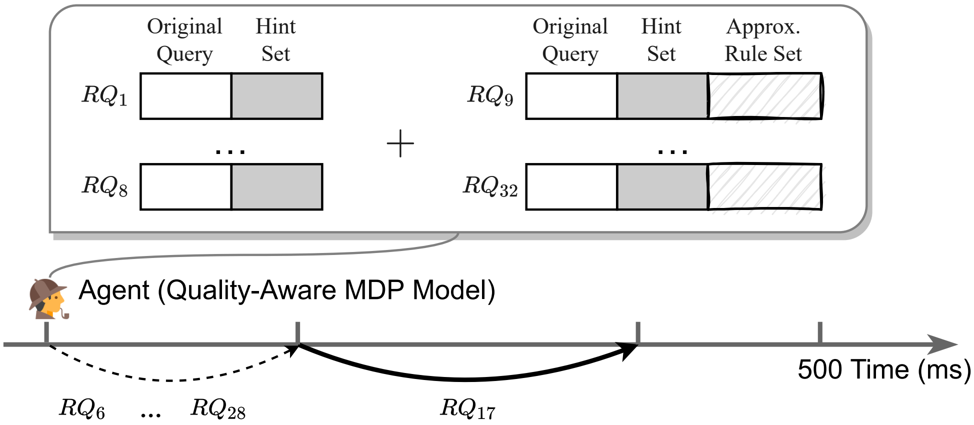


Figure 2.9: One-stage MDP approach.

One-stage approach. A natural idea is to replace the MDP model in Section 2.4 with the quality-aware MDP model. We let the MDP agent simultaneously consider query hints and approximation rules as rewriting options. By applying the new reward function combining both the efficiency of the rewritten query and the result's quality, the MDP agent learns to maximize the chance of generating viable rewritten queries and maximize the quality.

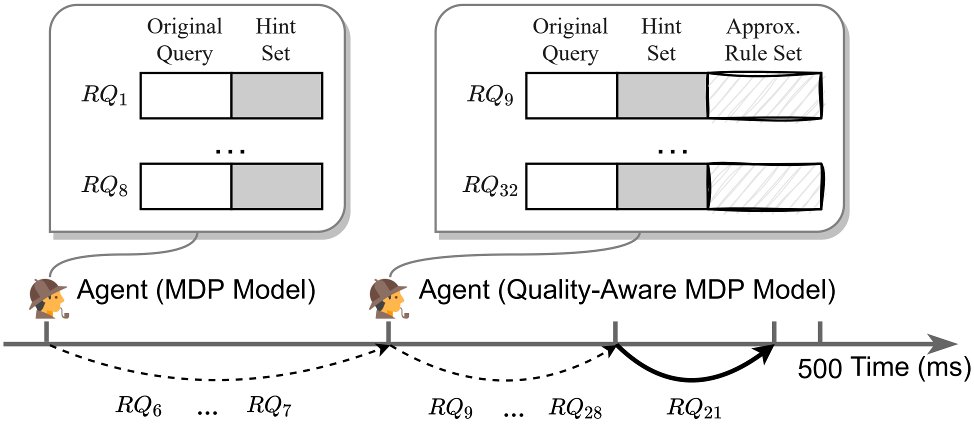


Figure 2.10: Two-stage MDP approach. After running the original agent that considers the 8 query-hint sets defined in Figure 2.4 without approximation rules, we cannot find a viable RQ. We then run the new agent with the quality-aware MDP model that considers all 8 query-hint sets combined with 3 approximation-rule sets (e.g., substituting the `tweets` table with 20%, 40%, or 80% sample tables), resulting in 24 rewritten queries in total. After spending extra time exploring a few RQs, the quality-aware agent chooses RQ_{21} as the final decision.

Two-stage approach. A drawback of the previous approach is that the agent might miss a non-approximate viable rewritten query. To solve this problem, we consider a two-stage approach, with a main idea to let the MDP agent exhaust all candidate query hints first and then explore those approximation rules. In the two-stage approach, Maliva first runs the original MDP model, excluding the approximation rules. If the agent finds a viable rewritten query, it outputs the RQ as before. If the agent exhausts all candidate RQs without finding a viable one, and the elapsed time has not exceeded the time budget τ, then we run the new quality-aware MDP model that considers the approximation rules to find a viable RQ.

When the planning time for the original agent is longer than the time budget, the two-stage approach reduces to the case described in Section 2.4. In this case, the one-stage approach is preferred since it can increase the chance of generating a viable rewritten query considering approximation rules. When the planning time for the original agent is relatively small compared to the time budget, the two-stage approach has the advantage of not missing any non-approximate viable rewritten queries.

2.6.3 Differences between Maliva and Bao

The recent Bao technique [62] also uses hints to rewrite queries. Maliva is closely related to Bao but different at multiple levels. First, to select a potentially viable query plan from all the candidate query-hint sets, Bao takes a brute-force approach by enumerating all options (using QTE). Maliva, in contrast, trains an MDP agent that explores the options by carefully balancing the planning time and query execution time. This difference makes Bao's method inapplicable in many visualization problems in the main application domain of Maliva. Secondly, as we will demonstrate in Section 2.7.6, when the number of candidate rewriting options is large (e.g., > 16), the planning time of Bao can exceed the time budget. Maliva, on the other hand, has a significantly shorter planning time and thus is capable of generating much more viable rewritten queries. Lastly, Bao does not consider approximate rewrites. Maliva, in contrast, offers flexibility by allowing approximate rewriting queries with minimal quality loss.

2.7 Experiments

We conducted experiments to evaluate Maliva[1]. In particular, we want to answer the following questions: (1) How well does it rewrite queries to support visualization requests? (2) How well does it generalize to different numbers of rewriting options? (3) How well does it perform for different types of queries (e.g. single-table selection queries and multiple-table joining queries)? (4) How well does it generalize to different time budgets, unseen queries and other

[1] Maliva is open-sourced on Github (`https://github.com/baiqiushi/maliva`)

databases? (5) How does it compare with related solutions? and (6) How much is its training overhead?

2.7.1 Setup

Datasets. We used two real datasets and a synthetic one as shown in Table 2.1. The Twitter dataset included 100 million geo-located tweets in the US from November 2015 to January 2017. We kept the timestamp, geo-coordinate, text message, and several user attributes for each tweet in a `tweets` table. For the experiment on join queries, we used the `tweets` table and a `users` table. The former had a foreign key of "user_id" referencing the "id" in the latter. We used the geo-coordinate attribute as the output for visualization (e.g., choropleth map, heatmap, or scatterplot). The NYC Taxi dataset [72] included taxi trip records within three years from 2010 to 2012. The third dataset was generated from the TPC-H benchmark [113]. We used the line-item table as the fact table. The attributes we used for query selection conditions are shown in Table 2.1.

Table 2.1: Datasets.

Dataset	**Record #**	**Size**	**Filtering Attributes**
Twitter	100,000,000	57GB	text, created_at, coordinates, users_statues_count, users_followers_count
NYC Taxi	500,412,914	146GB	pickup_datetime, trip_distance, pickup_coordinates
TPC-H	300,005,811	65GB	extended_price, ship_date, receipt_date

Query workloads. We generated random queries on each dataset for training and evaluation. Take Twitter dataset as an example. We first randomly sampled a set of tweets from the base table. For each tweet, we generated a query as follows. We chose the `text`, `created_at`, and `coordinates` attributes for the selection conditions in the query. We generated three conditions based on the values in the sampled tweet. For `text`, we randomly selected a non-stop word in the original tweet's text message as the keyword condition. For `created_at`, we generated a temporal range condition with the value in the original tweet as the left boundary. We divided the maximum range on the `created_at` attribute in the base table into multiple zoom levels, and randomly selected a level to generate the length of the range condition. Suppose the maximum range on `created_at` had L days. We computed the maximum zoom level on `created_at` as $Z = \lceil log_2(L) \rceil$. If we randomly chose a zoom level from range $[0, Z]$ as z, we computed the length of the query condition range as $l = max(L/2^z, 1)$. Similarly, for the `coordinates` attribute, we used the exact coordinates in the sampled tweet as the center. We randomly chose a zoom level and generated a spatial bounding box as the spatial range condition for the query.

In the experiments, we divided the queries into three disjoint sets: a training set, a validation set and an evaluation set. We used a hold-out validation strategy to choose the best agent. When evaluating different approaches, the "difficulty" of the queries in the evaluation workload played an important role. That is, if none of the physical plans of a query were viable, then no approach can generate a viable plan without approximation. On the contrary, if a high percentage (e.g., over 50%) of the physical plans are viable, it would be easy for any method — even a trivial one that picks plans at random — to find a viable plan. In this regard, we further divided the evaluation workload into subsets of queries based on their difficulty measured by the number of viable plans. In our evaluation, we focus on "difficult" queries where less than 50% plans are viable since they can better distinguish the performance of different methods.

QTE implementations. We implemented two QTEs to evaluate the Maliva's performance. 1) *Accurate-QTE*. To isolate the effect of estimation errors on the Maliva' performance, we used the actual execution time of the hinted queries as the estimation, and set up a unit cost parameter to represent the time of collecting the selectivity value of one filtering condition in a given rewritten query. Unless otherwise stated, we used $40ms$ as the unit cost of collecting one selectivity value for the *Accurate-QTE*. 2) We also implemented the ML-based *approximate-QTE* as presented in Section 2.4.2. We used a random sample table [124] to estimate the selectivity values of query conditions. The selectivity values were used by the approximate-QTE's ML model to estimate the execution time of queries.

Performance metrics. We used two metrics to evaluate the performance of different approaches. Recall that a generated rewritten query is "*viable*" if its total response time (including both the planning time and the querying time) is within a given time budget. The "viable query percentage" (VQP) of a solution was the ratio of viable queries over all the queries in the workload. The other metric was called "*Average Query Response Time*" (AQRT), which was the average total response time of all the queries in the workload.

Query-rewriting Approaches. We compared the proposed MDP-based approaches with three related methods, i.e., baseline, naive, and Bao [61]. MDP-based approaches included an MDP agent using an approximate-QTE, i.e., MDP (Approximate-QTE), and an MDP agent using an accurate QTE, i.e., MDP (Accurate-QTE). In the baseline approach, the middleware relies on the database optimizer to generate a physical plan for the original query. In the naive approach, we used the same approximate QTE as the MDP-based approach, but enumerated all possible RQs in a brute-force way, then chose the best RQ as the output. The third approach was Bao [61]. We used its open-source release [11] as the server, which provided interfaces for training the model and using the model to choose the best plan for a given set of query plans. Its original client, which was a PostgreSQL plug-in, did not support query hints for using a specific index, which were required by our visualization queries. To solve this problem, we implemented a new client in Python to support such query hints while keeping their server implementation.

In the experiments, we ran both the database and the middleware on the same AWS t2.xlarge instance with four vCPUs, 16GB RAM, and a 500GB SSD drive. We implemented the middleware in Python 3.6 and the neural network using Pytorch 1.7. We evaluated Maliva on both PostgreSQL and a commercial database. All figures were results on PostgreSQL if not stated otherwise.

2.7.2 Performance on Using Query Hints

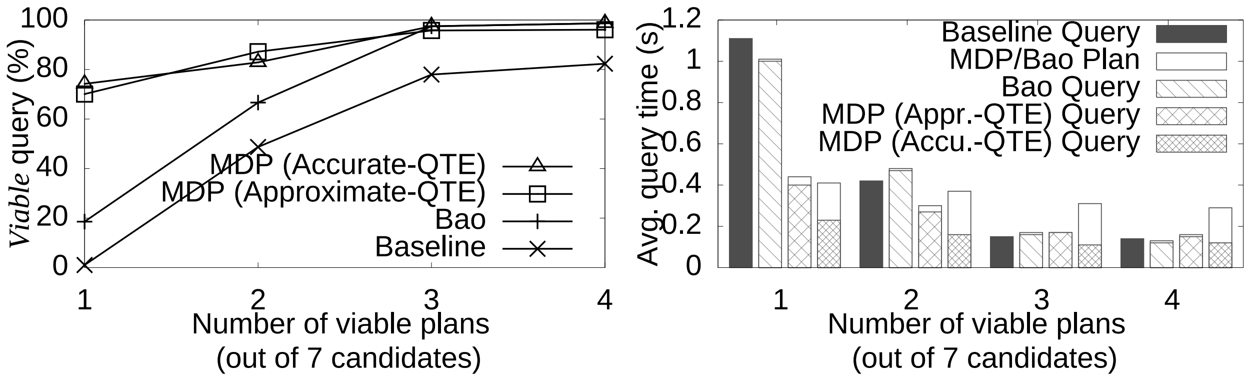
We evaluated the performance of Maliva for only considering query hints in rewriting options (i.e., no approximations). For each dataset, we generated queries with three filtering conditions and set up the rewrite-option set with 8 query-hint sets, i.e., using or not using the index on each attribute. Since one of the 8 hint sets was "no hint at all", which was the original query, the total number of candidate physical plans was 7, i.e., the original query's physical plan was one of the 7 hinted queries. We varied the evaluation workloads with different numbers of viable plans (i.e., $1-4$ out of 7), and collected the VQP and AQRT metrics for each approach. Table 2.2 shows the number of queries in the evaluation workloads.

Table 2.2: Number of queries in evaluation workloads.

# of viable plans	0	1	2	3	4	≥ 5
Twitter	518	97	234	118	153	69
NYC Taxi	408	91	146	13	181	3
TPC-H	381	107	310	66	47	0

Figures 2.11(a), 2.12(a), and 2.13(a) show viable-query percentages (VQP) on the three datasets. The MDP-based approaches and Bao outperformed the baseline approach significantly, with MDP (Accurate-QTE) as the best. For example, on the Twitter dataset, for the queries with a single viable plan, both MDP-based approaches increased the VQP from the baseline's 1% and Bao's 20% to more than 70%. In most cases, MDP (Approximate-QTE) performed better than or comparable to Bao. In one case of the TPC-H dataset, Bao performed better than MDP (Approximate-QTE) mainly because Bao's QTE had a much higher accuracy than the approximate QTE for TPC-H. When the number of viable plans increased from 1 to 4, the VQP of all approaches increased because the more viable plans existed for a query, the easier it was for each approach to find a viable plan in a short amount of time.

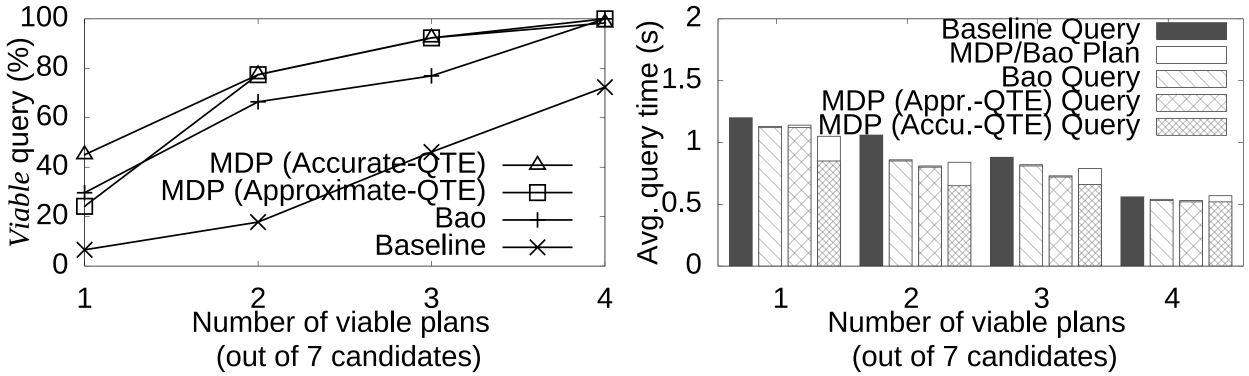
Figures 2.11(b), 2.12(b), and 2.12(b) show the results of the average query-response time



(a) Viable query percentage.

(b) Avg. query response time.

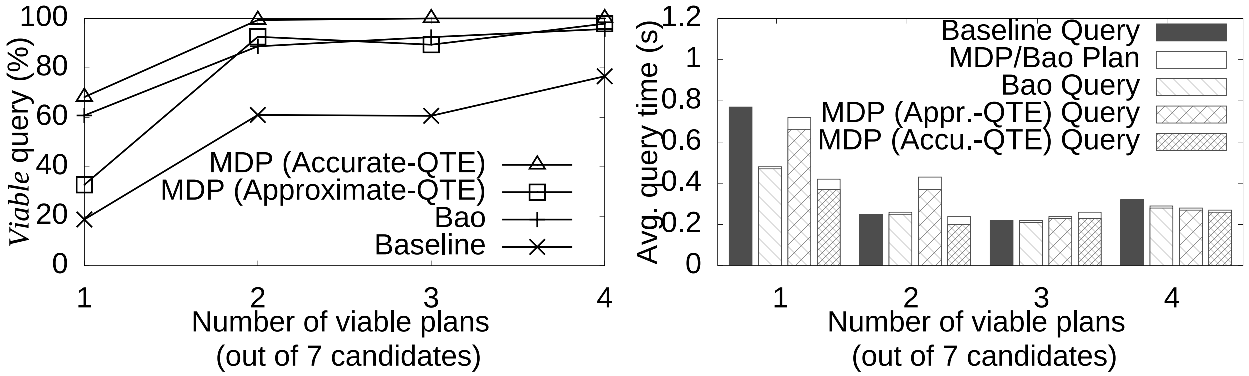
Figure 2.11: Performance on the Twitter dataset ($\tau = 500ms$).



(a) Viable query percentage.

(b) Avg. query response time.

Figure 2.12: Performance on the NYC Taxi dataset ($\tau = 1s$).



(a) Viable query percentage.

(b) Avg. query response time.

Figure 2.13: Performance on the TPC-H dataset ($\tau = 500ms$).

(AQRT) of different approaches. On the Twitter dataset, Bao had a comparable AQRT to the baseline, while MDP (Approximate-QTE) had much lower time than the baseline and Bao for queries with one or two viable plans. For example, MDP (Approximate-QTE) reduced the average response time from the baseline's 1.11 seconds and Bao's 1.01 seconds to 0.4 seconds. On the NYC Taxi dataset, Bao and MDP-based approaches had comparable performance and were slightly better than the baseline. On the TPC-H dataset, Bao was better than or comparable to the baseline. In two cases, Bao performed better than MDP (Approximate-QTE) because Bao's QTE had a much higher accuracy than the approximate QTE on TPC-H. However, in all cases, MDP (Accurate-QTE) always had a lower query time than Bao and the baseline, which means it generated a more efficient plan. In cases where MDP (Accurate-QTE) had a longer response time, the extra planning time was the main reason. At the same time, the high VQP of MDP (Accurate-QTE) proved the ability of the MDP model balancing the planning time and the query-execution time to maximize the chance of generating a viable rewritten query.

2.7.3 Effect of Rewrite-Option Number

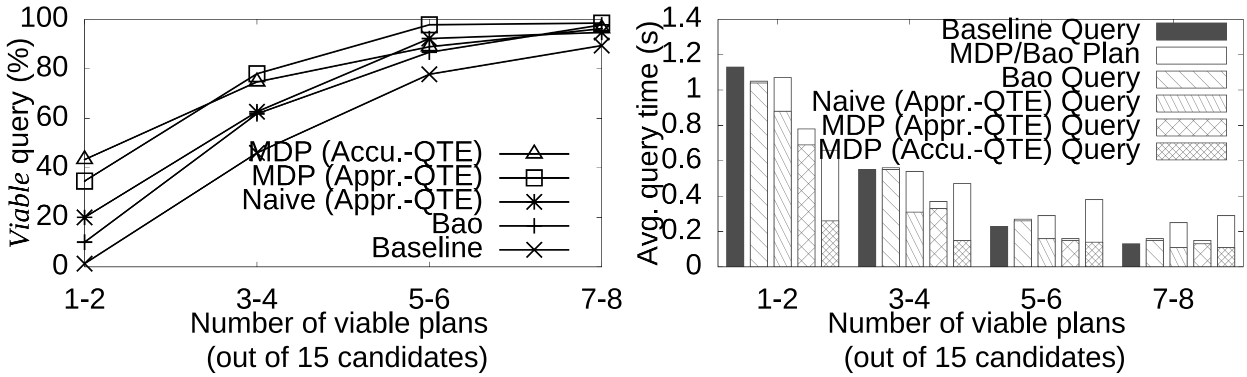
We evaluated the effect of the number of rewriting options on the Twitter dataset. We set up workloads of queries with different numbers of filtering conditions, resulting in different numbers of rewriting options. To illustrate the planning efficiency of MDP-based approaches, we also evaluated a naive approach, i.e., Naive (Approximate-QTE), which enumerated all possible RQs, estimated their time using the approximate QTE, and chose the best RQ as output. Table 2.3 shows the number of queries for the workloads.

Table 2.3: Workloads with 16 rewriting options.

# of viable plans	0	1-2	3-4	5-6	7-8	≥ 9
# of queries	485	150	241	90	132	93

As shown in Figure 2.14(a), the two MDP approaches performed the best, generating up to

40× more viable queries than both Bao and the baseline approach on queries with one or two viable plans.



(a) Viable query percentage.

(b) Avg. query response time.

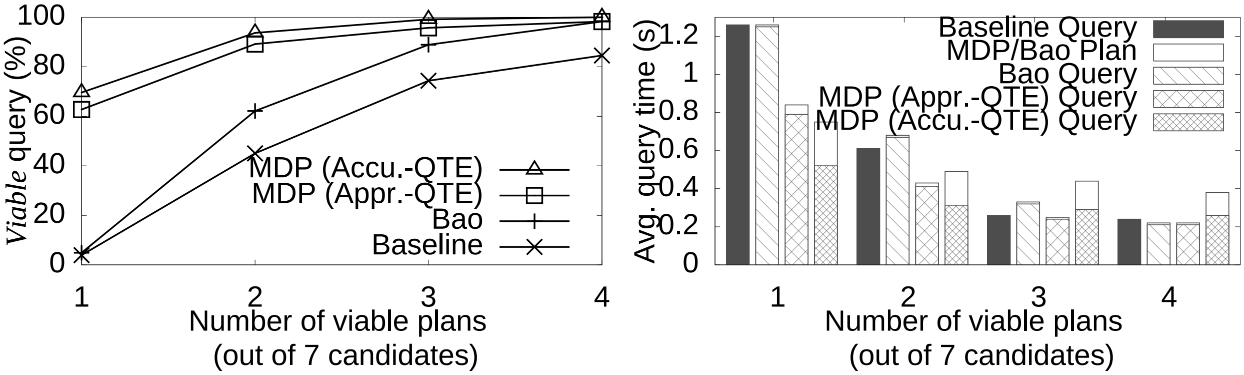
Figure 2.14: Performance for 16 ROs on the Twitter dataset ($\tau = 500ms$).

Figure 2.14(b) shows the AQRT results. Consistent with the VQP results, MDP-based approaches outperformed both Bao and the baseline approach. For example, MDP (Approximate-QTE) reduced the average response time from the baseline's 1.13 seconds and Bao's 1.05 seconds to 0.66 seconds for queries with one or two viable plans. Note that in both VQP and AQRT results, the MDP-based approach performed significantly better than the naive approach using the same approximate QTE. These results show the benefit of MDP-based careful planning strategy over a brute-force enumeration approach.

2.7.4 Effect of Time Budget

We evaluated the effect of time budget on the performance of different approaches. We varied the time budget on the Twitter dataset. We show results for 1-second time budget.

As shown in Figure 2.15(a) and (b), the MDP-based approaches outperformed both Bao and the baseline approach significantly. MDP (Accurate-QTE) outperformed MDP (Approximate-QTE) since the agent could afford the expensive estimation cost for more accurate estimations to find better-rewritten queries. These results show that the MDP model is adaptive



(a) Viable query percentage.

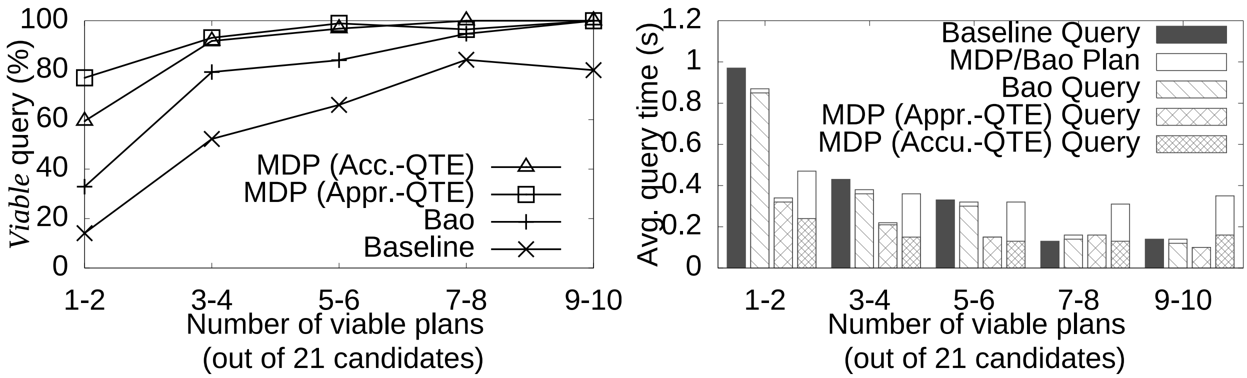
(b) Avg. query response time.

Figure 2.15: Performance for 1-second time budget on the Twitter dataset.

to QTEs with different costs and accuracies for different time budgets. Compared with the results in Figure 2.11 where the time budget was $500ms$, MDP (Accurate-QTE) performed better when the budget was higher, and MDP (Approximate-QTE) performed better when the budget was lower.

2.7.5 Performance on Join Queries

To evaluate the performance of Maliva on queries with joins, we set up a workload of queries joining the `tweets` and `users` tables with filtering conditions on three attributes. For the MDP-based approaches and Bao, we considered 7 different ways of using or not using indexes on the three attributes and 3 different join methods (i.e., nest-loop-join, hash-join, and merge-join) between the two tables. Thus we had 21 query-hint sets in total as the rewriting options. Figure 2.16(a) shows that for all workloads, the MDP-based approaches outperformed Bao. For the queries with only one or two viable plans, MDP (Approximate-QTE) generated more than twice as many viable plans as Bao. Figure 2.16(b) shows that MDP (Approximate-QTE) outperformed Bao in all cases. For queries with one or two viable plans, the MDP-based approach reduced the average query response time from Bao's 0.87 second to 0.34 second.



(a) Viable query percentage.

(b) Avg. query response time.

Figure 2.16: Performance for join queries (Twitter dataset, $\tau = 500ms$).

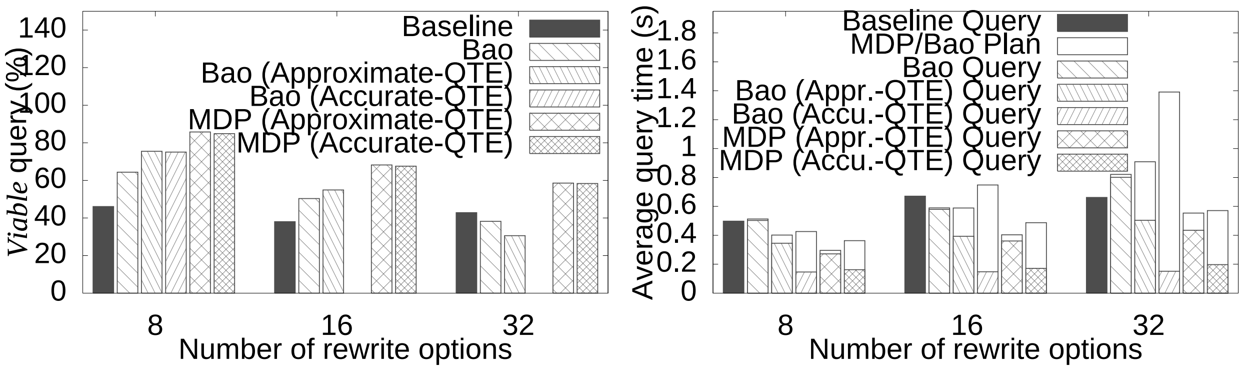
2.7.6 Additional Comparison with Bao

We further compared the performance of Maliva with Bao to demonstrate the advantage of our approach (see Figure 2.17). Besides the original Bao approach, we included two additional variants — *Bao (Approximate-QTE)* and *Bao (Accurate-QTE)* — that integrated Bao's enumeration strategy on top of our QTEs. We focused on "difficult" queries where less than 50% physical plans were viable. We used the Twitter dataset and varied the number of rewrite options from 8 to 32 (as described in Section 2.7.3).

Table 2.4: Workloads of queries where less than 50% plans were viable.

# of rewrite options	8	16	32
# of queries	449	481	497

As shown in Figure 2.17, our MDP-based approaches outperformed both the baseline approach and Bao-based approaches significantly in all the cases. For the 8 rewrite-option workload, both Bao-based approaches using the approximate-QTE and the accurate-QTE outperformed the original Bao. The reason was that Bao's own QTE relied on the plan tree and operators' cost estimations from the physical plan generated by PostgreSQL. As a result, it suffered from the significant estimation errors by PostgreSQL for textual and spatial filtering conditions. With the help of the approximate and accurate QTEs' more accurate estimations, the performance of Bao was improved. However, when the number



(a) Viable query percentage.

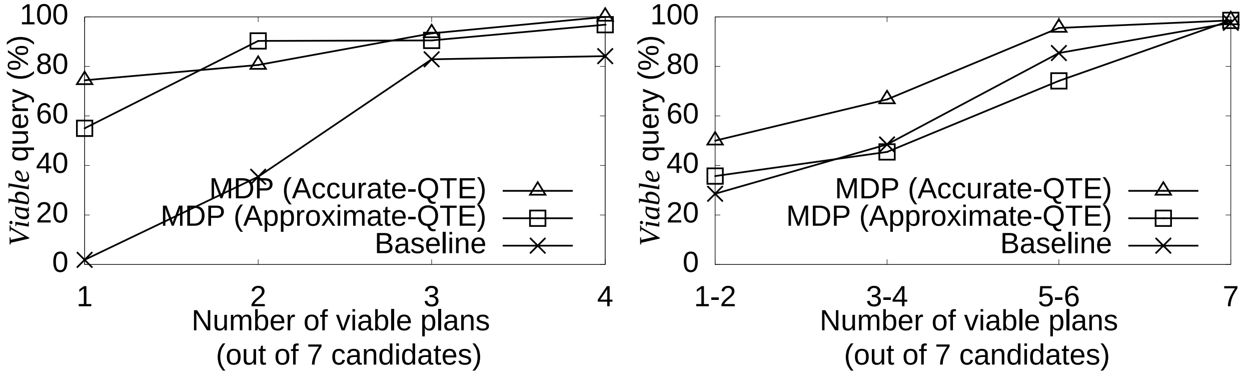
(b) Avg. query response time.

Figure 2.17: Comparison with Bao on the Twitter dataset ($\tau = 500ms$).

of rewrite options was 32, both Bao (Approximate-QTE) and Bao (Accurate-QTE) performed even worse than the baseline due to the high cost of estimating all the candidate plans in the brute-force query-planning phase. The VQP of Bao (Accurate-QTE) dropped to 0% because the planning time exceeded the $500ms$ time budget. As shown in the 32 rewrite-option column of Figure 2.17(b), by judiciously choosing which rewritten queries to run the expensive accurate-QTE, the MDP (Accurate-QTE) reduced the average planning time from Bao (Accurate-QTE)'s 1.24 seconds to 0.37 seconds, with a reduction of more than 70%. This result showed the superiority of using the MDP-based approach for query planning over Bao's brute-force approach.

2.7.7 Unseen Queries and Other Databases

To evaluate how well Maliva can be generalized to handle unseen queries, we did experiments on the Twitter dataset to train and test the MDP model using two workloads with different query shapes. The training queries were on a single `tweets` table with three filtering conditions (textual, temporal, and geospatial). In comparison, the testing queries joined the `tweets` table and the `users` table on `user_id` with three filtering conditions (textual, temporal, and geospatial) on the former table. As shown in Figure 2.18(a), the MDP-based approaches outperformed the baseline significantly on the workload with unseen queries. For example, for queries with a single viable plan, the MDP (Approximate-QTE) approach increased the VQP from the baseline's 2% to 55%, and the MDP (Accurate-QTE) approach further increased it to 74%.



(a) Unseen queries ($\tau = 500ms$).

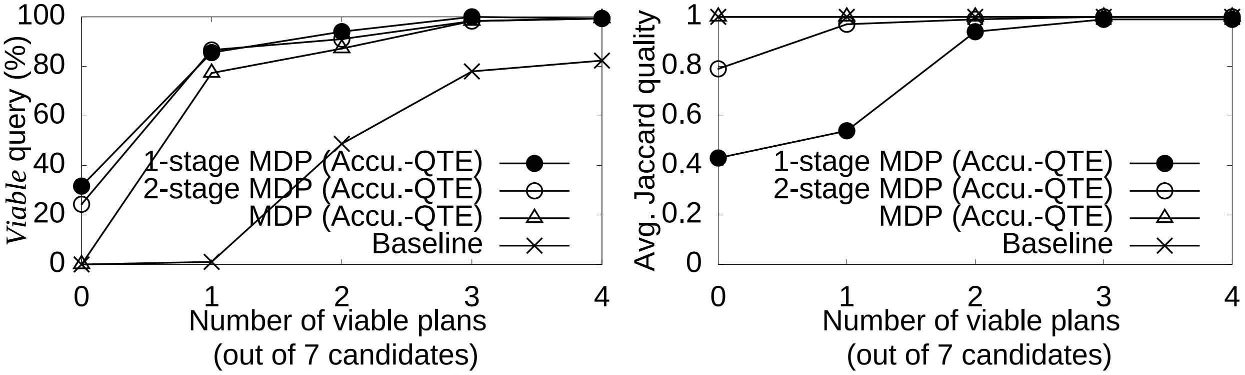
(b) Commercial DB ($\tau = 250ms$).

Figure 2.18: Generalization to (a) handle unseen queries and (b) use a commercial database.

We also did experiments on the Twitter dataset using a commercial database. We used a smaller table with 10 million records and thus a smaller time budget ($250ms$). The result is shown in Figure2.18(b). Due to the commercial database's complex behaviors, the approximate QTE had a much lower accuracy (two orders of magnitude) than it had on PostgreSQL. The reason was the approximate QTE only considered predicates' selectivities for estimation, but more factors in the commercial database affected the query time, such as buffering and dynamic execution plan change. However, MDP (Approximate QTE) still had comparable performance (VQP) to the baseline. With a more accurate yet more expensive QTE, MDP (Accurate-QTE) outperformed the baseline for all the queries. For example, for queries with one or two viable plans, the baseline had a VQP of 23%, MDP (approximate-QTE) had a VQP of 36%, and MDP (Accurate-QTE) increased the VQP to 50%.

2.7.8 Performance of Quality-Aware Rewriting

We evaluated the performance of the two quality-aware query rewriting approaches (i.e., one-stage and two-stage) described in Section 2.6. We used the same Twitter dataset and workload as in Section 2.7.2. We compared them with the baseline approach and the MDP approach without considering approximation rules. For the quality-aware rewriting approaches, we considered five approximation rules (i.e., adding a `LIMIT` clause with 0.032%, 0.16%, 0.8%, 4%, and 20% of the estimated cardinality of the query) in addition to the eight query-hint sets considered in Section 2.7.2. All MDP approaches used an accurate-QTE. Besides the AQP and AQRT metrics, we collected a new metric called *Jaccard-based Quality*, which computed the Jaccard similarity between the visualization result of a rewritten query and that of the original query.



(a) Viable query percentage.

(b) Avg. Jaccard-based quality.

Figure 2.19: Performance of quality-aware rewriting (Twitter, $\tau = 500ms$).

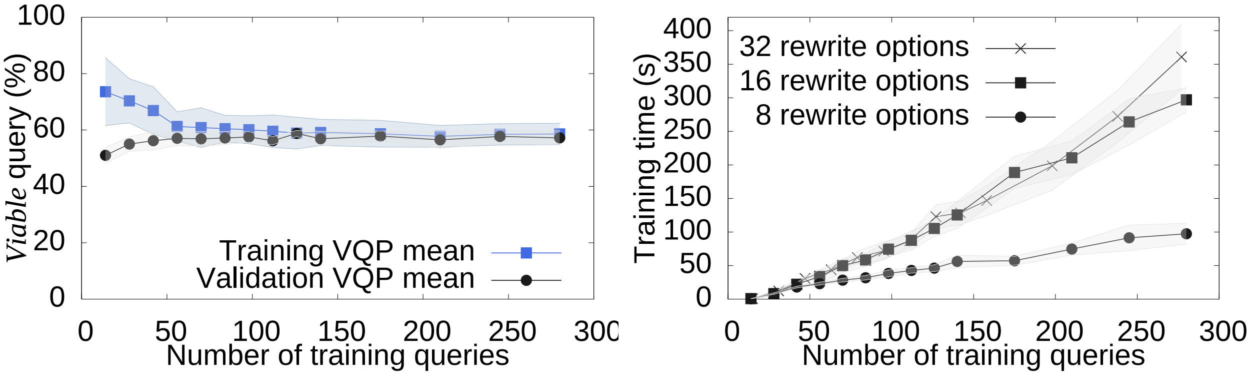
Figure 2.19(a) shows the VQP of these approaches. For the group of queries without any viable plan, the MDP approach without considering approximation rules and the baseline approach had a zero VQP. By generating approximate rewritten queries, the two-stage MDP approach increased the VQP to 24%, and the one-stage MDP approach further increased the VQP to 31%. There were 518 queries in the 0-viable-plan workload (Table 2.2), and the one-stage MDP approach generated more than 35 viable queries than the two-stage approach. In terms of efficiency, the one-stage MDP approach outperformed the two-stage approach in all cases. Figure 2.19(b) shows the average Jaccard-based quality of the rewritten queries generated by different approaches. Both the baseline and the MDP approach without considering approximation rules had no quality loss. The two-stage MDP approach had a significant advantage over the one-stage approach in terms of quality. For example, The former increased the quality of the 0-viable-plan queries from the one-stage approach's 0.43 to 0.79.

2.7.9 Training Performance

We evaluated the training performance for workloads with different numbers of rewriting options on the Twitter dataset. For each workload, we divided a set of about $1,400$ queries into a training set and a validation set. Then we varied the number of training queries and randomly sampled those from the training set without replacement. We then used the sampled queries to train an MDP agent and tested its performance on both the training queries and the validation queries. We repeated the step ten times for each number of training queries and collected the mean and standard deviation of the VQPs. We conducted the experiments on the MDP approach using the Accurate-QTE. We show results for 8 rewriting options.

Figure 2.20(a) shows the trend when we varied the number of training queries. The VQP on the validation set was close to the VQP on the training set for about 50 training queries. Figure 2.20(b) shows the training time of different numbers of rewrite options on the training sizes. For the same number of training queries, more rewrite options resulted in a larger q-network, which took more time to update the weights. For the workload with thirty-two rewrite options, it took about 150 seconds to train an MDP agent on 150 training queries.

Remarks: The experiments show that Maliva outperformed the baseline in terms of both the number of viable queries and average query response time. Maliva generated up to



(a) Learning curve for 8 ROs.

(b) Training time for different # of ROs.

Figure 2.20: Learning curve and training time on the Twitter dataset. The shaded area is plotted with "mean + standard deviation" as the upper bound and "mean − standard deviation" as the lower bound.

70× more viable queries than the baseline. The advantages of Maliva were shown in both the real and synthetic datasets, for different numbers of rewriting options, time budgets and query workloads. Its offline training overhead was relatively small. By considering approximation rules, Maliva generated even more viable queries. The comparison with Bao shows the advantage of Maliva due to the fact these two techniques were designed with different settings and optimization goals.

Limitations: One limitation of Maliva is that when the number of rewriting options was significant (e.g., ≥ 32), both the training and the online planning overhead of the MDP models became expensive. Also, for different sets of rewriting options, Maliva requires training different models.

2.8 Conclusions

In this chapter, we studied how to rewrite database queries to improve execution performance in middleware-based visualization systems. We explored two optimization options of adding hints and doing approximation. We developed a novel solution called Maliva, which adopts a Markov Decision Process (MDP) model to rewrite a visualization request under a tight time constraint. We gave a full specification of the solution, including how to construct an MDP model, how to train an agent, and how to use approximating rewriting options. Our experiments on both real and synthetic datasets showed that Maliva performed significantly better than the baseline without no-rewriting options in terms of both the probability of serving a visualization request within a time budget and query execution time.

Chapter 3

Supporting Human-Centered Query Rewriting in Middleware

3.1 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 [28, 49, 81], one family of technique is query rewriting, which transforms a query to a new query that computes the same answers with a higher performance.

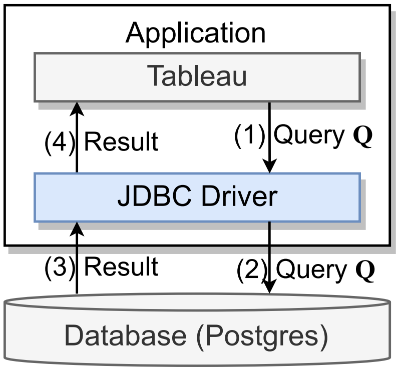


Figure 3.1: Query lifecycle between Tableau and Postgres.

Motivating example. Figure 3.1 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 3.2a 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 3.2: 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 3.2b, is equivalent to Q, and uses an `ILIKE` predicate. This expression can be answered using a trigram index on the `content` attribute [85], 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 [94].

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 [83] to introduce a rewriting rule inside the database, but as we will show in Section 3.2, 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 3.7.5, 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 3.2 and our experiments using different applications and databases on both synthetic and real-world datasets in Section 3.7.5 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 its source code is not available; and ii) the source code of the application is too complicated or old to modify, especially for legacy systems [48]. For example, reports [103] 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, certain cardinality constraints, known content properties, or primary keys and foreign keys), even though these rules may not be valid for all databases. Specifically, the experimental results in Section 3.7.5 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 chapter and the following chapter, we propose QueryBooster, a novel middleware-based service 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. In this chapter, we focus on the core technical aspects of the QueryBooster system. QueryBooster provides an expressive and easy-to-use rule language (called VarSQL) for users (SQL developers or DBAs) to define rewriting rules. 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 solution, 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 chapter, we study these challenges and make the following contributions.

- We propose a novel middleware-based query rewriting solution to fulfill the need of human-centered query rewriting (Section 3.3).
- 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 3.4).
- 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 3.5).
- We present a framework to search the candidate rewriting rules efficiently and suggest high-quality rules based on user-given examples (Section 3.6).
- 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 3.7).

3.2 Limitations of Existing Solutions

In this section, we show why existing solutions cannot solve the formulated problem. Figure 3.3 gives an overview of various solutions for query rewriting in the lifecycle of a query in a database system [5, 22, 62, 121, 131] and the position of the proposed QueryBooster solution. At a high level, these solutions can be classified into two categories: native writing plugins and third-party solutions.

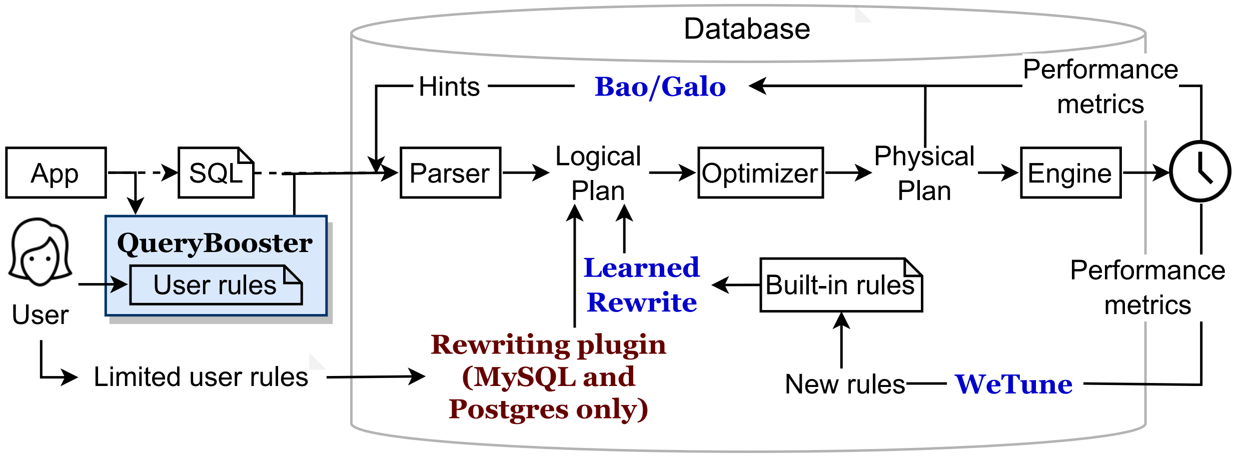


Figure 3.3: Query-rewriting solutions for databases (native solutions in brown and third-party solutions in blue).

Native rewriting plugins. Most databases such as AsterixDB [4], IBM DB2 [41], MongoDB [66], MS SQL Server [104], MySQL [70], Oracle [74], Postgres [83], SAP HANA [99], Snowflake [102], and Teradata [91], 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 (i.e., the SQL code), 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 [83]. Its rule language cannot express the rewriting in the running example in Figure 3.2. 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 [84] 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 [70]. 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 [62] and Galo [22] 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 [121] generates new rewriting rules automatically by searching the logical-plan space and considering the performance of rewritten queries. LearnedRewrite [131] 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 [39] 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 [47] 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 [27] uses AI/ML techniques to recommend rewriting ideas for queries on MySQL and Postgres. Other systems such as ApexSQL [8], Query Performance Insights for Azure SQL [90], and Toad [111] 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 [79, 1] 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.

3.3 QueryBooster: A Human-Centered Query Rewriting Solution

In this section, we present a novel middleware solution called QueryBooster, to fulfill the need for human-centered query rewriting.

Figure 4.2 shows the query lifecycle using QueryBooster to rewrite application queries. It includes two phases, an offline *rule formulation* phase and an online *query rewriting* phase.

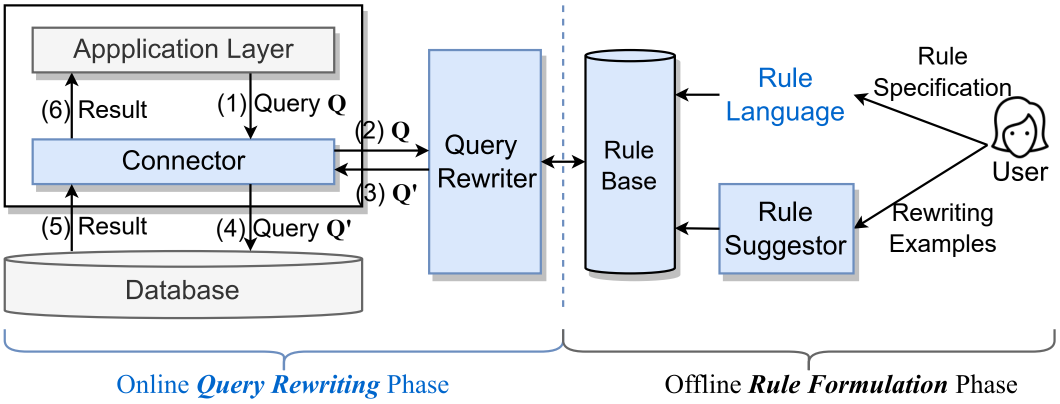


Figure 3.4: Query lifecycle of using QueryBooster to rewrite application queries.

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 solution, 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. In this Chapter, we focus on the technical aspects of the solution. We will discuss more how the connector is implemented and the license considerations in Chapter 4, where we focus on the system aspects of the proposed solution. To implement such a powerful solution, we have the following two tasks.

(Task 1) Developing an expressive and easy-to-use rule language. The first task of QueryBooster is to provide an expressive and easy-to-use rule language for users to formulate rules. It should 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, or a column is unique in the table schema. *(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. Providing a rule language that meets the aforementioned three requirements is challenging. In Section 3.4 we first study the suitability of existing rule languages in the literature and then develop a novel rule language that combines advantages from existing languages to meet all the requirements.

(Task 2) Suggesting rules from examples. The second task of QueryBooster is to provide a rule-suggestion framework that automatically generalizes user-given rewriting examples into rewriting rules and suggests high-quality ones. Manually formulating a rewriting rule

that covers many queries is tedious. We want a better experience in which a user expresses the rewriting intention by providing query rewriting pairs. Then, the system can automatically suggest rewriting rules to achieve the rewritings of the given examples. For instance, if a user inputs the rewriting pair in Figure 3.2 as one of the examples, the rule suggestor can automatically generalize it and recommend the following rule to the user.
STRPOS(**LOWER**(<x>), '<y>') --> <x> **ILIKE** '%<y>%'

Developing such a rule suggestor is also challenging since we need to answer a few questions, such as how to measure the quality of rewriting rules, how to generalize query rewriting pairs into rewriting rules, and how to search the candidate rewriting rules to suggest high-quality ones. We answer them in Sections 3.5 and 3.6.

Correctness of rewriting rules. In the case where users make mistakes when formulating rewriting rules, we can leverage existing query equivalence verifiers (e.g., [18, 121]) to validate the rules and guarantee their correctness.

3.4 VarSQL: A Rewriting-Rule Language

In this section, we focus on providing an expressive and easy-to-use rule language for QueryBooster's users to formulate rewriting rules that work for different applications and databases. 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.

3.4.1 Suitability of Existing Rule Languages

We study existing rule languages and their suitability for QueryBooster. These languages [33, 100, 28, 81, 23, 17, 82, 123, 19, 5] are summarized in Figure 3.5. We categorize the languages in two aspects: general versus SQL-specific and declarative versus imperative, and then summarize how the languages in different categories meet the requirements of the rule language of QueryBooster.

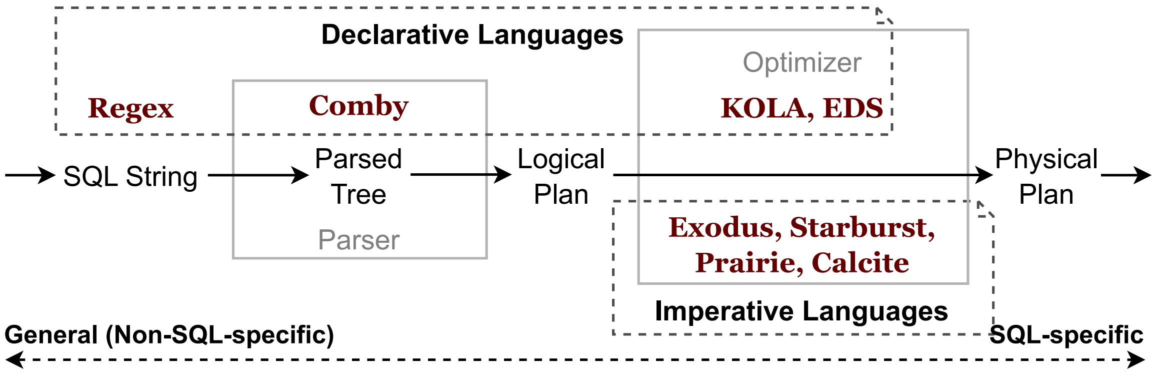
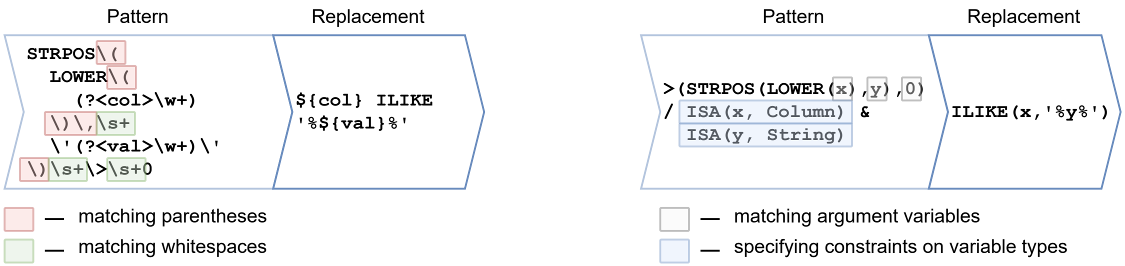


Figure 3.5: Existing rule languages (shown in brown) in the lifecycle of a SQL query.

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, regular expressions [123] (shown as "Regex") do not require the input string to be a SQL query. On the contrary, rule languages on the right such as EDS [28] only accept valid SQL query plans as the input.

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, to achieve the rewriting of replacing "`STRPOS`" functions with "`ILIKE`" predicates, we can write a rule using either regex as shown in Figure 3.6a, or EDS as shown in Figure 3.6b. Compared to regex, the EDS language has two advantages. First, we do not need to specify SQL-specific syntax requirements such as white spaces and arguments in functions. Second, we can specify SQL-specific constraints for variables that are not supported by regex, such as "`x` is a column and `y` is a `String` literal."

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



```
STRPOS\(
  LOWER\(
    (?<col>\w+)
  \)\,\s+
  \'(?<val>\w+)\'
\)\s+\>\s+0
```

```
${col} ILIKE
'%${val}%'
```

(a) A rule written in regex.

```
>(STRPOS(LOWER(x),y),0)
/ ISA(x, Column) &
  ISA(y, String)
```

```
ILIKE(x,'%y%')
```

□ — matching argument variables
🟦 — specifying constraints on variable types

(b) A rule written in the EDS language [28].

Figure 3.6: Rewriting rules to replace STRPOS functions with ILIKE predicates in two different languages.

system (called "EDBMS") and its SQL dialect [28]. Second, they require the users to deeply understand how a SQL query is translated into a plan and how the database optimizer works. For example, to formulate the rule in the EDS language shown in Figure 3.6b, a user has to translate the original SQL predicate of "STRPOS(· · ·) > 0" into a logical plan tree format with ">" as the parent node, which can be counter-intuitive for end users who are familiar with the SQL syntax but not a database engine.

Declarative versus Imperative. The existing rule languages are either *declarative* or *imperative*. In a declarative rule language, users describe how a rule (e.g., pattern and replacement) looks, but not how a rule should be implemented. On the contrary, in an imperative rule language, users specify a sequence of steps that should be taken to do the rewriting. For example, regex is declarative. Calcite [5] is imperative, and it uses Java to formulate a rule. 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. Thus, we need schema information from the database to implement this rule. Using an imperative rule engine, we can easily write a rule with a few lines of C code [82] that accesses the schema data and checks the matching condition.

To this end, we summarize how existing languages meet QueryBooster's requirements on its rule language in Table 3.1. An observation is that no existing rule language satisfies all the requirements. Next, we develop a novel rewriting-rule language called VarSQL.

Table 3.1: Suitability of existing languages for QueryBooster.

Rule Language	**Expressive Power**		**Independent of DB**	**Additional Knowledge Needed**
	SQL Semantics	*SQL Schema*		
Regex	No	No	**Yes**	
Comby	No	No	**Yes**	
KOLA	**Yes**	No	**Yes**	① ② ③
EDS	**Yes**	**Yes**	No	① ②
Exodus	**Yes**	**Yes**	No	① ②
Starburst	**Yes**	**Yes**	No	① ② ④ ⑤
Prairie	**Yes**	**Yes**	No	① ② ⑤
Calcite	**Yes**	**Yes**	**Yes**	① ② ④ ⑥
VarSQL	**Yes**	**Yes**	**Yes**	

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

3.4.2 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.

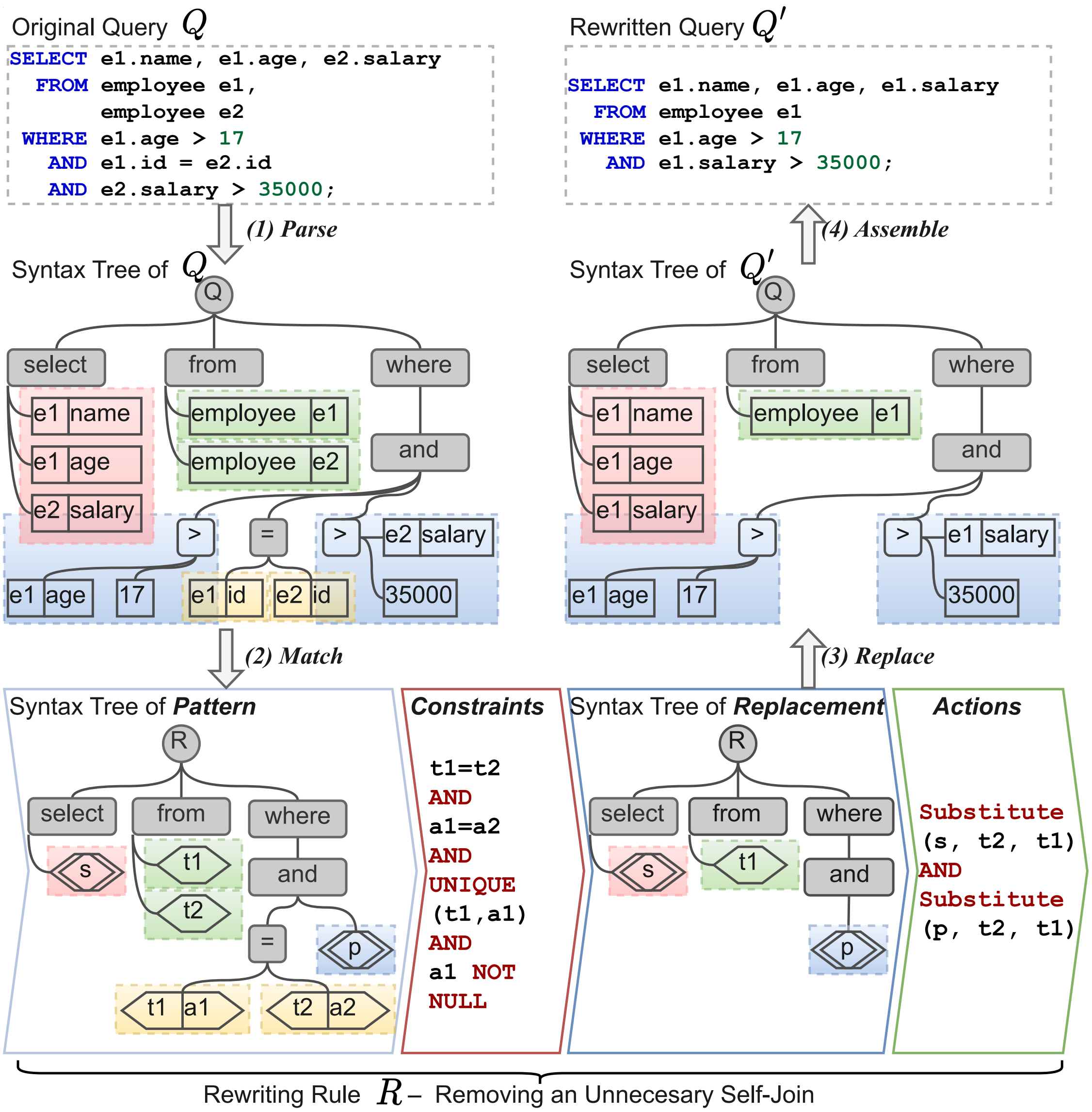
The syntax of VarSQL to define a rewriting rule is as follows:
[**Rule**] ::= [**Pattern**] / [**Constraints**] --> [**Replacement**] / [**Actions**].

[1] VarSQL stands for “Variablized SQL”.

VarSQL uses a four-component structure adopted by most rule languages (e.g., EDS [28] and Comby [19]). 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 3.2.

To minimize the learning cost for end-users to define rules in VarSQL, we introduce only two variables, namely "element-variable" and "set-variable." An element-variable can match any individual element in a SQL query, such as a table, a column, etc. A set-variable can match any collection of elements in a SQL query, such as the column list in the `SELECT` clause. Note that keywords and delimiters cannot be represented as variables. An entire clause cannot be represented as any type of variable either. For example, a set-variable can match all the columns in the `SELECT` clause, but no type of variable can match the entire `SELECT` clause.



Original Query Q

```
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;
```

Rewritten Query Q'

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

Constraints

```
t1=t2
AND
a1=a2
AND
UNIQUE
(t1,a1)
AND
a1 NOT
NULL
```

Actions

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

Figure 3.7: 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 Q's elements.

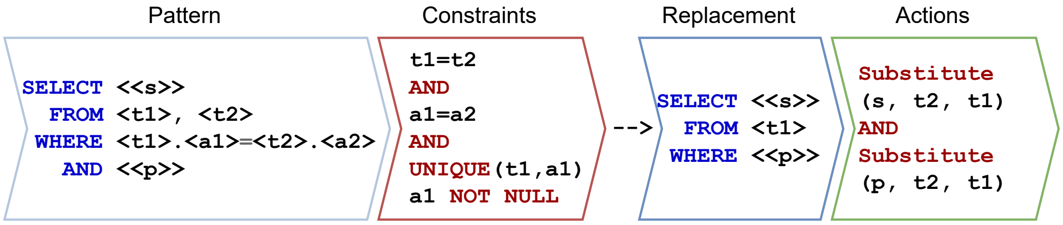
Table 3.2: Variable definitions in VarSQL.

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 syntax tree level. Consider the rule R shown at the bottom of Figure 3.7, where the pattern and replacement are 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 3.7 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. For completeness, we also show the string representation of the rule R in Figure 3.8.

Providing pre-implemented imperative procedures. Based on SQL, VarSQL is a



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

```
t1=t2
AND
a1=a2
AND
UNIQUE(t1,a1)
a1 NOT NULL
```

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

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

Figure 3.8: An example rule in VarSQL that removes an unnecessary self-join.

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 3.8 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 3.8 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 3.7 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.

3.4.3 VarSQL-Based Rewriting Engine

In this section, we discuss how to design an extensible rewriting engine to support the VarSQL rule language. First, rewriting SQL queries using VarSQL-based rules can happen at the string level, abstract syntax tree (AST) level, or the logical plan tree level. We will discuss the pros and cons of implementing the engine at different levels and why we choose the AST level. Second, we discuss a few problems in implementing the engine, such as the termination of the rewriting process and the uniqueness of the rewriting results. Finally, we briefly discuss how the rewriting engine can support different databases and SQL dialects.

Order-insensitive rewriting at the abstract syntax tree (AST) level. VarSQL operates the pattern matching and replacing at the abstract syntax tree level, other than general languages such as Regex and Comby that operate at the string level or SQL-specific languages such as EDS and KOLA that operate at the query logical plan level. A major problem of operating at the string level is that the order of elements in the pattern is important. For example, in Comby, if the pattern is defined as

```
SELECT orders.*, customers.* FROM orders, customers,
```

it does not match a query with a different order of joining the same two tables, such as

```
SELECT orders.*, customers.* FROM customers, orders.
```

This is counter-intuitive for SQL users because, in SQL semantics, the above two queries share the same pattern.

Operating at the logical plan level can solve the problem. However, we know that one SQL query can have many equivalent logical plan trees with different orders of joins and different positions of filtering predicates. Thus, it requires the rewriting engine to enumerate all possible logical plans for a specific query to do pattern-matching, which can be expensive. For instance, in Figure 3.9, all the three logical plans are possible output for a particular input query. In this case, checking a pattern matching against all possible logical plans of the query requires varying not only different join orders but also the positions of filtering predicates, which can make the implementation of the rewriting engine complex.

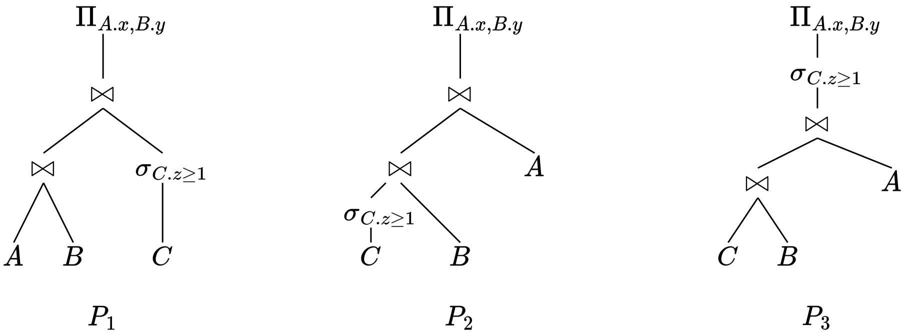


Figure 3.9: An example of three possible logical plans for a single SQL query which joins tables A, B, and C with a filtering predicate on the z column of C.

Thus, VarSQL operates at the AST level and does order-insensitive pattern-matching inside a clause to solve the problem. We have explained the pattern-matching process of a VarSQL-based rule on an example query in Section 3.4.2. We now focus on the order-insensitivity of the process. When a pattern matches a SQL query, VarSQL does not enforce the ordering of elements under a specific clause. Figure 3.8 shows a rule that specifies 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. This rule's pattern can match both queries shown in Figure 3.10, in which the join predicate (i.e., "`<t1>.<a1>=<t2>.<a2>`") and the predicate set-variable (i.e., "`<<p>>`") have different orders. This behavior is also consistent with the order-insensitive nature of SQL queries, where different orders of the same set of predicates under a `WHERE` clause should yield the same result.

Termination of the rewriting process and the uniqueness of the result. We have talked about how the rewriting engine does the pattern matching and replacing for given a SQL query and a rewriting rule. We now briefly discuss the entire rewriting process for

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

(a) An original query ($Q1$).

```
SELECT sum(o1.total_price)
  FROM orders o1,
       orders o2
 WHERE o1.os in ('ios', 'macos')
   AND o1.oid = o2.oid
   AND o2.state_code = 'CA';
```

(b) An original query ($Q2$).

Figure 3.10: Two original queries that both match the pattern of the rule in Figure 3.8.

an arbitrary online query. All user-formulated rules are stored in the rule base as shown in Figure 4.2. When an original query Q comes, the query rewriter accesses each rule r in the rule base and checks its pattern and constraints against query Q. If the rule r matches Q, the query rewriter rewrites Q to Q' according to the rule's replacement and actions. It then treats Q' as Q and repeats the rule matching and rewriting process until the query cannot match any rule in the rule base. Note that the same rule can be matched and used multiple times.

To ensure the process always terminates, the query rewriter tracks the rewriting path and discovers cycles by checking if the same rewritten query appears more than once on the path. It breaks a cycle by returning the repeated rewritten query as the result. One concern is that different orders of applying rules on a query may result in different rewritten queries. To address this concern, QueryBooster allows users to specify a priority for each rule so that high-priority rules can be applied first. For example, a user may ask the system to apply general optimization rules (e.g., removing unnecessary type-castings) before applying specific optimization rules (e.g., replacing substring functions).

Supporting different databases and SQL dialects. QueryBooster can support different databases and SQL dialects for two reasons. One is its simple design of extending SQL with variables. The other is that the query rewriter operates at the Abstract Syntax Tree (AST) level and adopts existing SQL parsers to parse the queries and rules. Existing rules are parsed into ASTs and then stored in the rule base. For an input query, the query rewriter first parses the query into an AST. It then does the pattern matching by comparing each rule's AST with the query's AST. Once a rule pattern matches the query, the query rewriter assembles the rewritten query's AST based on the replacement AST of the rule. Finally, it uses the same SQL parser to assemble the rewritten query's AST back to a SQL query.

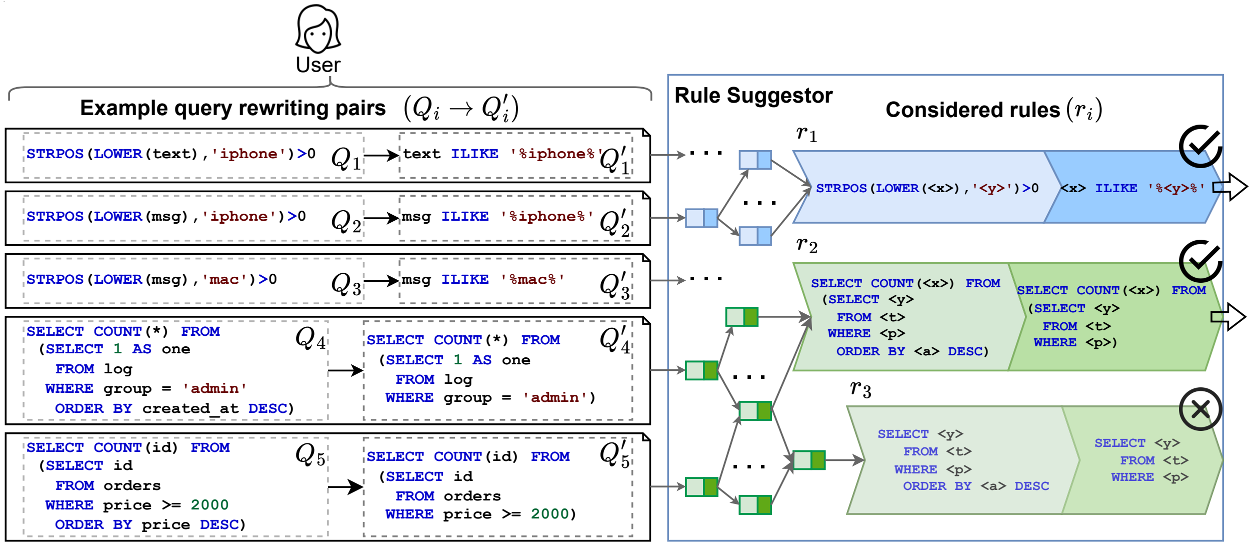
Thus, QueryBooster can be easily extended to support different databases with different SQL dialects. To support a new database and its SQL dialect, we replace the SQL parser inside the query rewriter with the corresponding SQL parser for the new database, and both the rule language and the query rewriting process can be adapted automatically.

3.5 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.

3.5.1 Quality of Rewriting Rules

Adopting MDL principle to measure the quality of 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 3.11. 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



```
STRPOS(LOWER(text),'iphone')>0        text ILIKE '%iphone%'
STRPOS(LOWER(msg),'iphone')>0         msg ILIKE '%iphone%'
STRPOS(LOWER(msg),'mac')>0            msg ILIKE '%mac%'
```

```
SELECT COUNT(*) FROM
 (SELECT 1 AS one
   FROM log
  WHERE group = 'admin'
   ORDER BY created_at DESC)
```

```
SELECT COUNT(*) FROM
 (SELECT 1 AS one
   FROM log
  WHERE group = 'admin')
```

```
SELECT COUNT(id) FROM
 (SELECT id
   FROM orders
  WHERE price >= 2000
   ORDER BY price DESC)
```

```
SELECT COUNT(id) FROM
 (SELECT id
   FROM orders
  WHERE price >= 2000)
```

```
STRPOS(LOWER(<x>),'<y>')>0        <x> ILIKE '%<y>%'
```

```
SELECT COUNT(<x>) FROM
 (SELECT <y>
   FROM <t>
  WHERE <p>
   ORDER BY <a> DESC)
```

```
SELECT COUNT(<x>) FROM
 (SELECT <y>
   FROM <t>
  WHERE <p>)
```

```
SELECT <y>
   FROM <t>
  WHERE <p>
   ORDER BY <a> DESC
```

```
SELECT <y>
   FROM <t>
  WHERE <p>
```

Figure 3.11: 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.

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 3.11, 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 [96], 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[30], data cleaning [93, 37], and regex learning [14]. 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 3.6.3.

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

Definition 3.1. *(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 3.2. *(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 $\{r_1\ , r_2\ , \ldots\ , r_k\}$ 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.

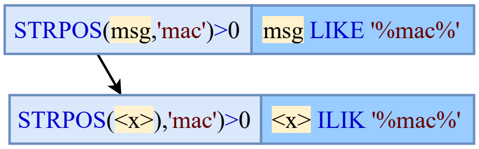
3.5.2 Transforming Rules to More General Forms

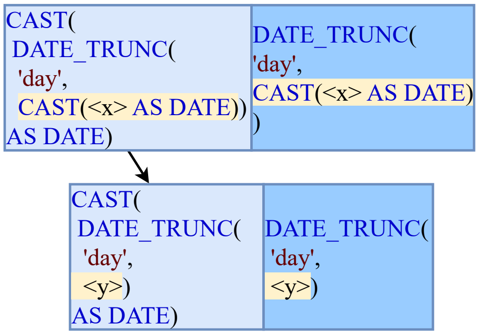
Transformations on rules. A transformation on a rewriting rule can generalize the rule into a more general rule such that the new rule covers more rewriting pairs than the original one. The instantiation of transformations is dependent on the given rule language. We now define transformations (shown in Figure 3.12) 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.

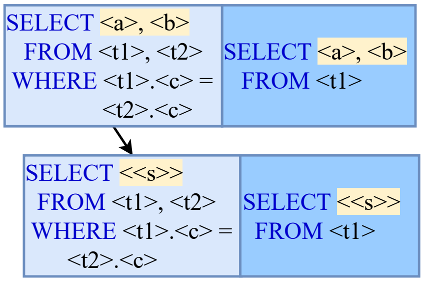
Variablize-a-Leaf. This transformation replaces an instantiated element (table, column, or value) in a rule with a variable. In this way, the transformed rule can match more queries than the original one. As shown in the first example in Figure 3.12, the transformed rule can match a query with any column 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 example in Figure 3.12, 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 3.12, 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

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>

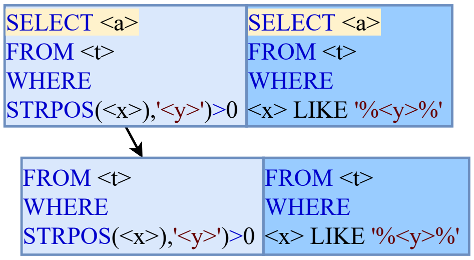
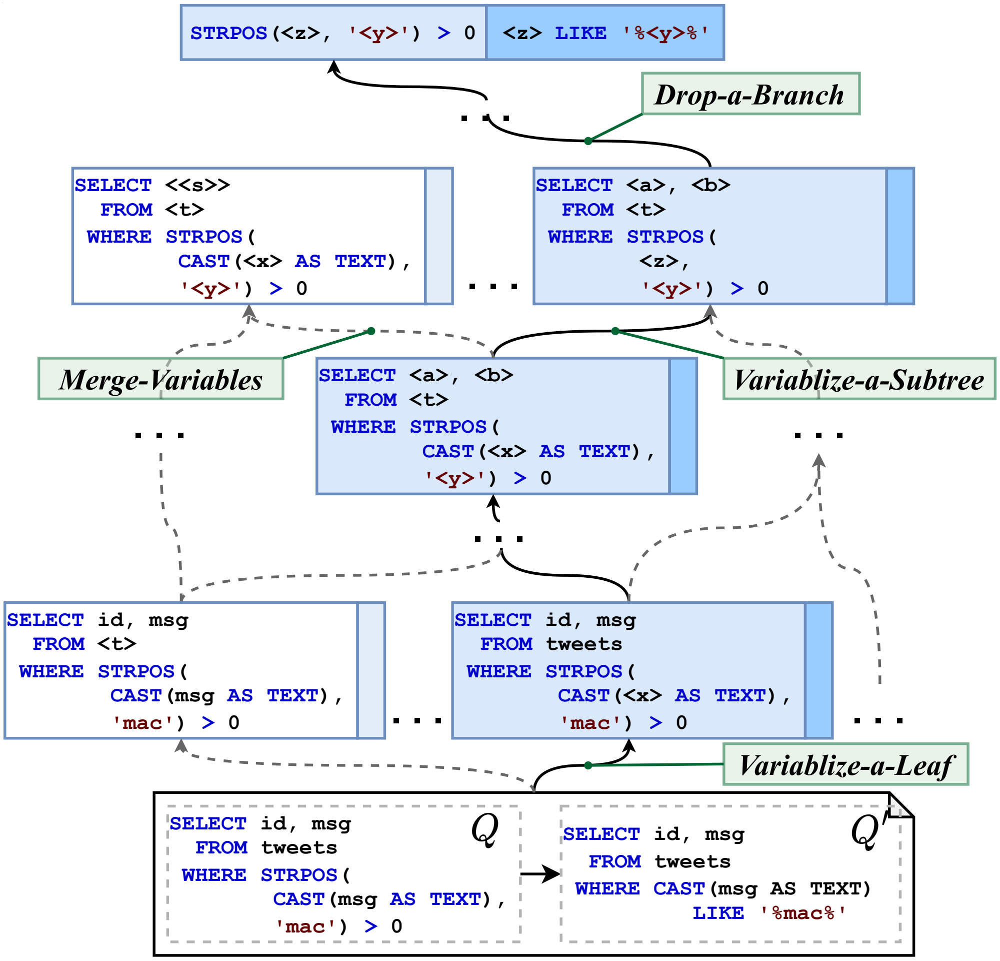
Drop-a-Branch	Remove a common branch (starting from the root) in both a rule's pattern and replacement ASTs.	
		SELECT <a> FROM <t> WHERE STRPOS(<x>),'<y>')>0
		SELECT <a> FROM <t> WHERE <x> LIKE '%<y>%'
		FROM <t> WHERE STRPOS(<x>),'<y>')>0
		FROM <t> WHERE <x> LIKE '%<y>%'

Figure 3.12: 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.



```
STRPOS(<z>, '<y>') > 0    <z> LIKE '%<y>%'
```

```
SELECT <<s>>
  FROM <t>
 WHERE STRPOS(
         CAST(<x> AS TEXT),
         '<y>') > 0
```

```
SELECT <a>, <b>
  FROM <t>
 WHERE STRPOS(
         <z>,
         '<y>') > 0
```

```
SELECT <a>, <b>
  FROM <t>
 WHERE STRPOS(
         CAST(<x> AS TEXT),
         '<y>') > 0
```

```
SELECT id, msg
  FROM <t>
 WHERE STRPOS(
         CAST(msg AS TEXT),
         'mac') > 0
```

```
SELECT id, msg
  FROM tweets
 WHERE STRPOS(
         CAST(<x> AS TEXT),
         'mac') > 0
```

```
SELECT id, msg
  FROM tweets
 WHERE STRPOS(
         CAST(msg AS TEXT),
         'mac') > 0
```

```
SELECT id, msg
  FROM tweets
 WHERE CAST(msg AS TEXT)
         LIKE '%mac%'
```

Figure 3.13: A rule graph generated from a given rewriting pair (Q, Q'). The vertices are generalized rules (only showing patterns due to the space limit). The solid edges show one path of generalizing the pair into a general rule. The green tags on the edges illustrate which transformations are applied.

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 3.12, 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 can gradually remove the irrelevant context of a rule's pattern from the top to the bottom of the pattern's AST.

Rule Graph. Until now, starting from an initial rewriting pair, by applying the transformations iteratively, we can generalize it into more and more general rewriting rules gradually. If we treat each newly-generated rewriting rule as a vertex and a transformation as an edge, we can obtain a graph of rewriting rules. We call it a *rule graph.* Figure 3.13 shows an example rule graph. As we can see, a rule graph for a single rewriting pair can be big, and the union of all rule graphs for a set of rewriting pairs can be even bigger, so searching for a set of high-quality rules is difficult. In the next section, we discuss how to navigate through the search space and make final suggestions to the users.

3.6 Searching For High-Quality Rules

To solve the rewriting-rule suggestion problem defined in Definition 3.2, we defined a set of transformations in Section 3.5.2 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 with the minimum description length. To solve the problem, we adopt a heuristic-based strategy to expand the candidate-rule set greedily and rely on a local set of rules to make final suggestions. In this section, we first present the greedy searching framework, then propose several heuristics to further reduce the search overhead.

3.6.1 A Greedy Searching Framework

We develop a method to search for rules, as shown in Algorithm 3. Its main idea is the following. 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 description 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).

The algorithm follows the hill-climbing paradigm [98], 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

Algorithm 3: A greedy algorithm for suggesting rules

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

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. As shown in the rule graph in Figure 3.13, 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.

3.6.2 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.

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 4 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.

Algorithm 4: m-promising-neighbor exploration

```
Input: A set of rewriting rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$
       A set of transformations $\mathcal{T} = \{T_1, T_2, \ldots, T_m\}$
       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 $\mathcal{C}$
$\mathcal{C} \leftarrow \mathcal{R}$
while $|\mathcal{C}| < m$ do
    // choose the most promising candidate rule
    $\hat{c} \leftarrow \arg\max_{c \in \mathcal{C}} \mathcal{P}(c)$
    // replace it with its 1-hop transformed child rules
    for $T_i \in \mathcal{T}$ do
        $T_i(\hat{c}) \leftarrow \{$all possible child rules by applying $T_i$ to $\hat{c}\}$
        $\mathcal{C} \leftarrow \mathcal{C} \cup T_i(\hat{c})$
    end
    $\mathcal{C} \leftarrow \mathcal{C} - \hat{c}$
end
return $\mathcal{C}$
```

Measuring the promisingness score of a rule. We consider three signals to measure a rule's promisingness score. First, one signal is the total length of those base rules that can be covered if we transform a candidate rule into a more general form. Second, another signal 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. This signal measures how far we can reach a more general rule starting from a particular candidate rule. Third, the last signal is the length of a candidate rule.

Formally, suppose we are given a set of base rewriting rules $\mathcal{R} = \{R_1, R_2, \ldots, R_n\}$ and a candidate rule c. We compute rule c's promisingness score $\mathcal{P}(c)$ as follows. For each base rule $R_i \in \mathcal{R}$, we compute a distance $\mathcal{D}(c, R_i)$, which is 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.

3.6.3 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 3), 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 $\mathcal{C}(\mathcal{W}_{\mathcal{R}})$. 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 $\Delta\mathcal{C}_c$. 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

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

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 3 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 4.

3.7 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?

3.7.1 Setup

Workloads. We used four workloads as shown in Table 3.3. 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 [121]. WeTune [121] is a technique that generates new rewriting rules automatically by searching the logical-plan space and considering the performance of rewritten queries. 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 [6].

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 [109] and Apache Superset [7] on top of both 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 [114]. We first inserted a 10GB TPC-H synthesized dataset into a PostgreSQL database (indexes were created using Dexter [2], which automatically creates indexes based on the database and workloads), 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 geo-spatial 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.

Table 3.3: 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)

Testbed. We implemented the QueryBooster solution using Python 3.9 and used the "mo-sql-parsing" package [52] 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 [93]. 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

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

3.7.2 A User Study to Evaluate Rule Languages

Table 3.4: 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 [123], and we used its C Sharp implementation provided by regex101 [95]. The other was the internal rule language used by WeTune [121], and we used its own implementation provided by its demo website WeRewriter [45]. 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.

Table 3.5: Results in the user study (% of users selected the rule language as the easiest to understand).

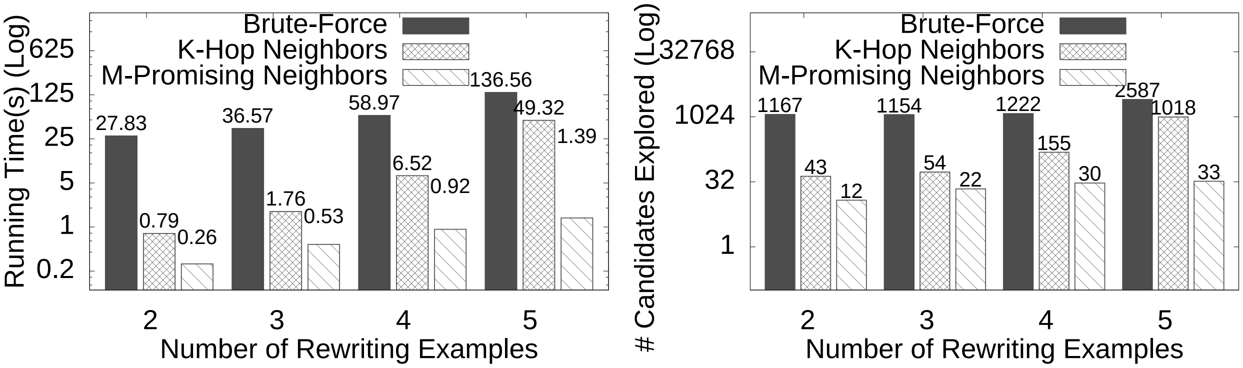
Pair Id	1	2	3
Workload	Twitter(Postgres)	Twitter(MySQL)	WeTune(Q91)
% of Regex	13.6%	4.5%	0%
% of WeTune	0%	13.6%	13.6%
% of VarSQL	**86.4%**	**81.8%**	**86.4%**

We invited 22 users who were familiar with SQL and with different backgrounds. The profiles of users are shown in Table 3.4, and the results are summarized in Table 3.5. 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.

3.7.3 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 3.6. 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 (Log-scale).

(b) Total # of candidates explored (Log-scale).

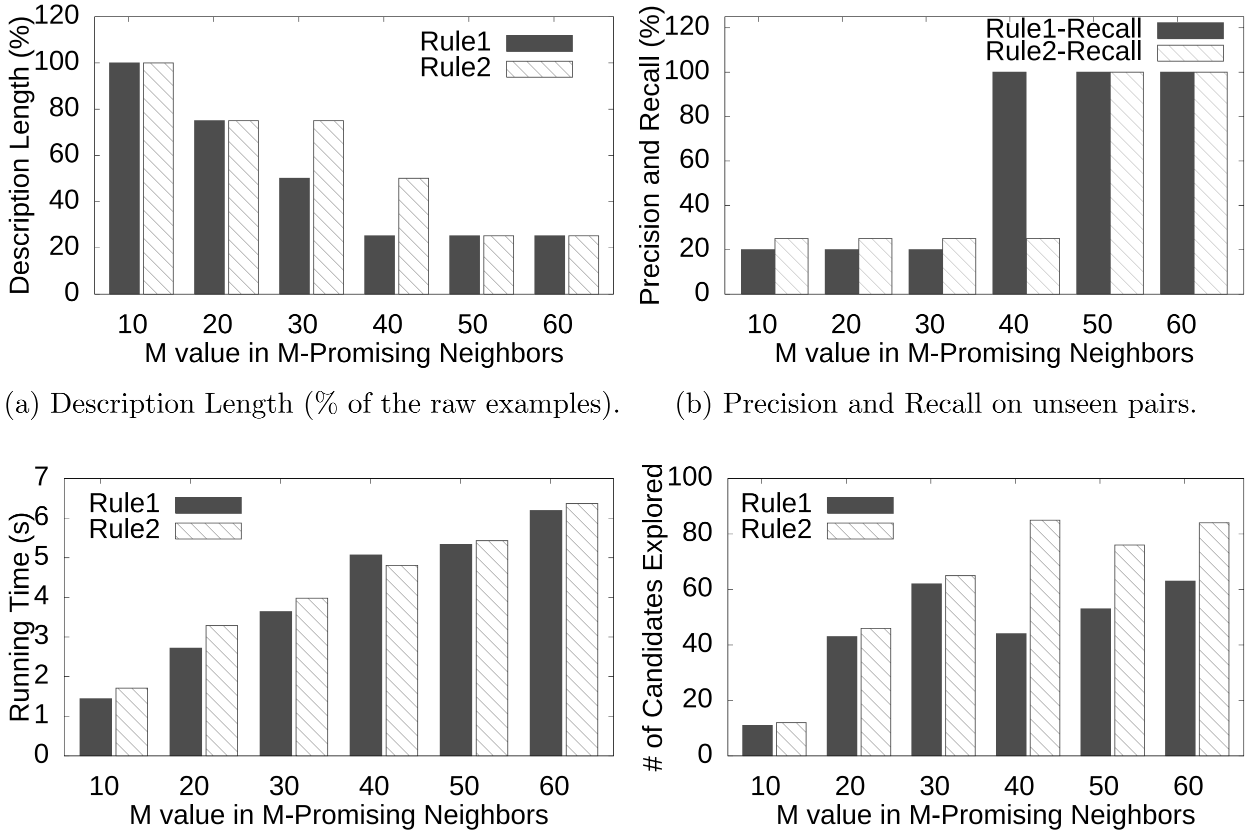
Figure 3.14: Comparison of different candidate exploration methods to suggest the same set of rules on the "Tableau + Twitter" workload.

The results are shown in Figure 3.14. As shown in Figure 3.14a, 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 3.14b, and both figures illustrate the correlation between the running time and the number of candidates explored in the searching framework.

3.7.4 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 3.15a. 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 3.15c, it only took about 5 to 6 seconds for the algorithm to output the optimal rule set. Figure 3.15d also shows the numbers of candidates explored for different m values.

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



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

(b) Precision and Recall on unseen pairs.

(c) Running Time (s).

(d) Total # of candidates explored.

Figure 3.15: Effect of the m value in the m-promising-neighbor searching strategy on the WeTune workload.

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 3.15b. 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. Thus, for this dataset, we recommend 50 for the m value.

3.7.5 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 [45] 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 physical plan.

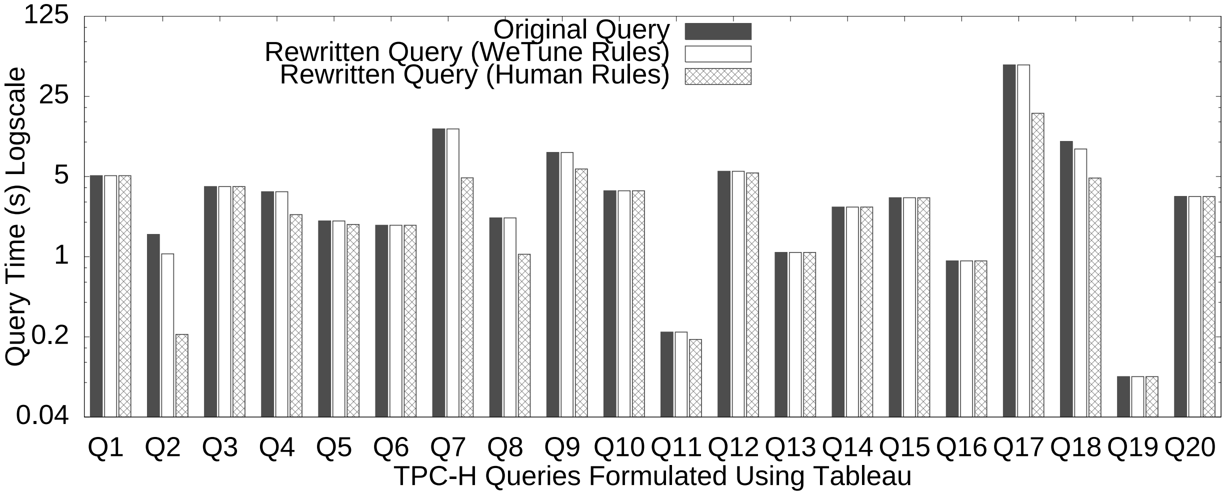
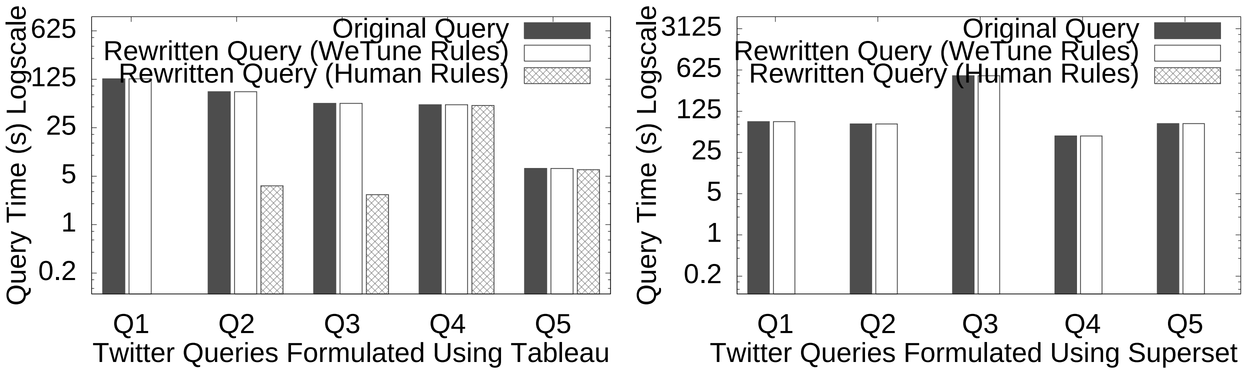


Figure 3.16: End-to-end query time (Log-scale) using QueryBooster to rewrite queries with WeTune-generated rules and human-crafted rules on "Tableau + TPC-H" workload compared to original query time in PostgreSQL.

Figure 3.16 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 3.17a and 3.17b 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 3.2. 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 3.17b, 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".

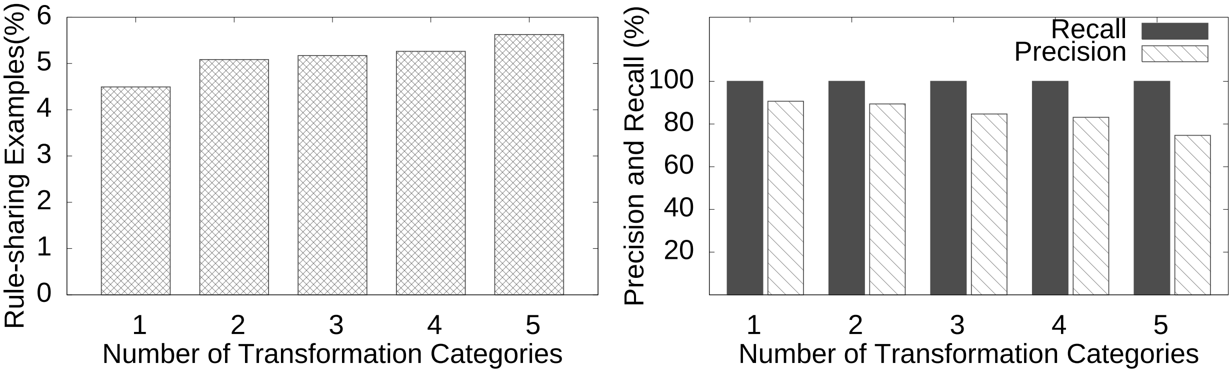
(b) "Superset + Twitter".

Figure 3.17: 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.

3.7.6 Generality of Rule Transformations

To evaluate the generality (covering more rewriting examples) of rule transformations, we used 178 rewriting pairs in the Calcite workload after removing some examples that the third-party SQL parser failed to parse. We applied the transformations defined in Section 3.5.2 to each rewriting example iteratively as long as they are applicable to generate more general rules. If multiple examples were generalized to the same rule, we only kept one copy of the rule. We divided the transformations into 5 categories: "*variablize-a-table*", "*variablize-a-column*", "*variablize-a-value*", "*variablize-a-subtree*", and "*merge-variables*", and we gradually increased the number of categories applied in the rule generalization process. We compared the rewriting result of the generated rules on the 178 examples. Since most rewriting pairs in the Calcite workload were designed for a unique rule, most of the examples were rewritten by the rules generated from themselves. 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 of the input examples.

Figure 3.18a shows that with more transformation categories used in generating rules, there were more examples rewritten by rules generated from other examples, which means the



(a) Rule sharing percentage.

(b) Precision and Recall.

Figure 3.18: The generality of rules generalized from the Calcite examples using different sets of transformations.

generated rules were more general. Figure 3.18b 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 the more transformations resulted in over-generalized rules that matched examples they should not match and rewrote the original queries to unintentional forms. This result also motivated our consideration of using MDL as an objective function to suggest rules, which prevented over-generalization for given examples.

3.7.7 Effect of Different Rule Quality Metrics

We also evaluated the effect of using different importance weights β in the *benefit* value (defined in Section 3.6.3) 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 3. 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.

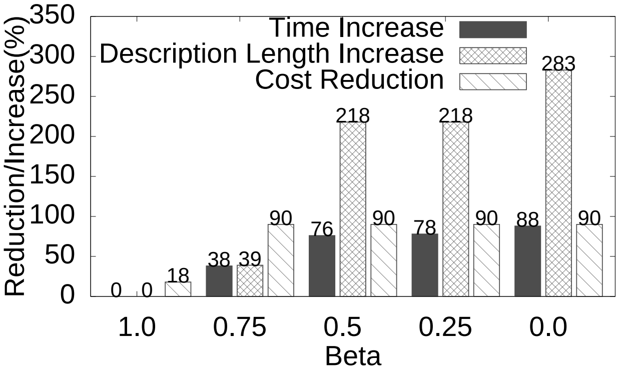


Figure 3.19: Effect of different β values.

The results are shown in Figure 3.19. When β was 1.0, the suggested rules only reduced the query cost by 18%. When β decreased to 0.75, 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%. Therefore, we recommend 0.75 as the β value for this dataset.

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%.

3.8 Conclusions

In this chapter, we proposed QueryBooster, a middleware solution 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.

Chapter 4

Supporting Middleware-Based SQL Query Rewriting as a Service

4.1 Introduction

System performance is critical in database applications where users need answers quickly to make timely decisions. SQL query rewriting [28, 81] is an optimization technique that transforms an original query to a rewritten one that computes the same answers with higher performance. Although query rewriting has been studied extensively as part of the query optimizer inside a database [28, 32], recent works [121, 10] have shown that purely relying on the rewritings inside traditional databases is insufficient to optimize modern query workloads. For instance, as previously discussed, with the prevalent use of business intelligence systems (e.g., Tableau and PowerBI), these machine-generated queries can be difficult for databases to optimize [121] and domain-specific knowledge and human-crafted rewriting rules are necessary to optimize workloads from different applications [88]. We can miss query rewriting opportunities (e.g., rewritings shown in Figure 4.7) due to various reasons. For instance, both the application and the database layers are black boxes and cannot be modified. Another reason is that existing rewriting plugins of databases have limited expressive power for users to express their rewriting needs.

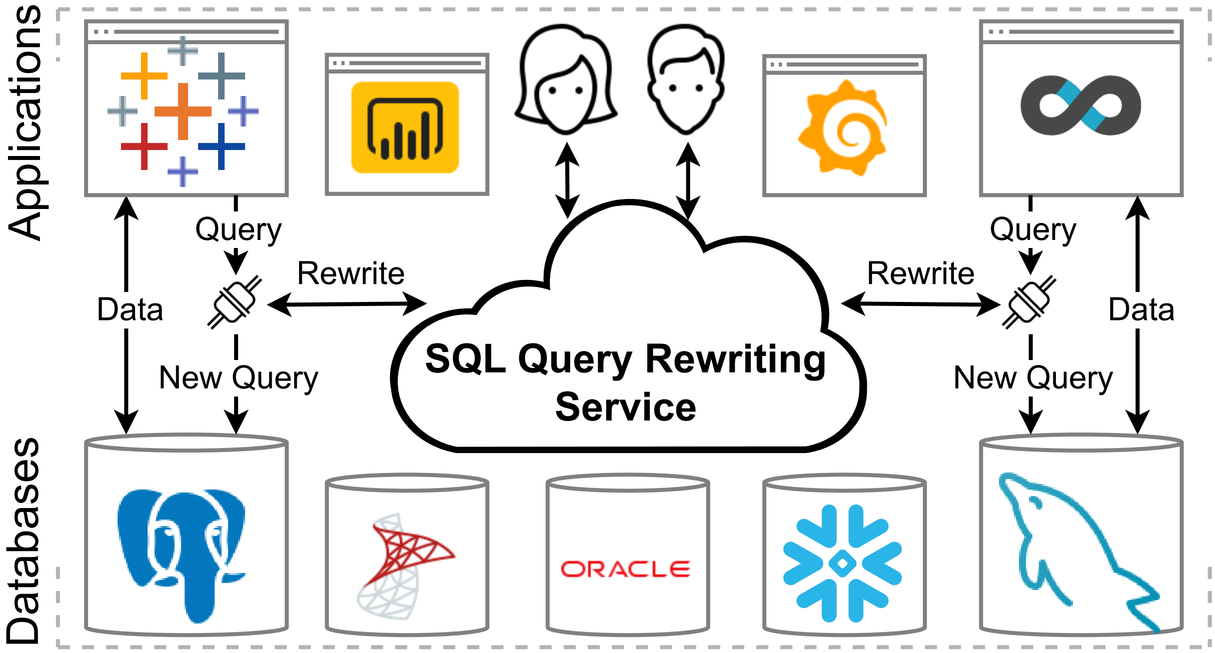


Figure 4.1: QueryBooster overview.

In this chapter, we detailed the design and architecture of QueryBooster, a middleware-based multi-user system to provide query rewriting between applications and databases as a service. The service intercepts and rewrites an original query from an application before it is sent to a backend database. It provides a web-based interface, using which customers manage their rewritings for different applications and databases. Users formulate rewriting rules using a language or by providing examples. They can see the statistics of different rewritings, such as the number of queries rewritten using a rewriting and the query's performance before and after the rewriting. Since rewriting SQL queries can be hard and time-consuming [131], instead of seeking advice from online forums (e.g., StackOverflow), users can also share and access rewriting rules using QueryBooster. The service requires no plugin installations to the user applications or databases. It also ensures high security as no user data goes through a third-party server. To summarize, QueryBooster has the following benefits. (1) It is easy to use, as users can use the interface to formulate, control, and monitor rewriting rules. (2) It enables users to share their rewriting knowledge and benefit from the wisdom of the crowd. (3) It is non-intrusive, as it requires no plugin installations to the user applications or databases.

Related Work: Most databases such as AsterixDB, IBM DB2, MS SQL Server, MySQL, Oracle, Postgres, Snowflake, and Teradata do not allow users to customize the rewritings of queries sent to the database. To our best knowledge, only Postgres and MySQL provide a plugin for users to define rules to rewrite queries before sending them to the database. However, their rule-definition languages have too limited expressive power, as shown in [88]. Commercial systems also do query rewriting for applications on top of databases. Keebo [47] uses machine learning and approximate query processing (AQP) techniques to accelerate analytical queries. It requires data to go through its server, which may introduce overhead and cause security concerns. EverSQL [27] uses AI techniques to recommend rewriting ideas for queries on MySQL and Postgres. Other systems such as ApexSQL [8] and Toad [111] help database developers analyze query performance bottlenecks. However, none of these solutions allow users to formulate their own rewriting rules to fulfill their rewriting needs. There are also service models such as database-as-a-service [35], function-as-a-service [106], etc. Compared to these systems, QueryBooster is the first system that supports SQL query rewriting as a service.

In the process of developing the QueryBooster system, we encountered many technical challenges and made a few decisions among different design choices. Next, we discuss those challenges and how we weigh the pros and cons of different approaches in designing each component. The remaining sections are organized as follows. We first introduce the system architecture of QueryBooster in Section 4.2. We use real-world applications and datasets to demonstrate the user experience of using QueryBooster to analyze application workloads, and introduce new rewriting rules to improve the queries performance and share rewriting knowledge among different users in Section 4.3. In Section 4.4, we discuss how to provide a user experience of managing and sharing rewritings intuitively and interactively. We discuss those design choices to minimize the intrusiveness of the QueryBooster system as a SQL query rewriting service in Section 4.5.

4.2 QueryBooster System Overview

Figure 4.2 shows the architecture of QueryBooster. Using the *Web UI*, users can log in to the system through the *User Manager* and manage applications through the *Application Manager*. They can have different sets of rewriting rules for different applications. Through the *Rule Manager* that integrates the technique proposed in Chapter 3, QueryBooster provides a powerful interface for users to formulate rewriting rules. It provides an expressive rule language (called VarSQL) for users to define rules. Users can easily express their rewriting needs by specifying the query pattern and its replacement. They can specify additional constraints and actions to express complex rewriting details. In addition, the service allows users to express their rewriting intentions by providing examples, each of which includes an original query and a rewritten one. QueryBooster automatically suggests high-quality rewriting rules based on the examples. The users can choose their desired rules and further modify suggested rules as they want. Rewriting rules are stored in the *Rule Base*.

QueryBooster provides database *Connectors* for users to download. Without any application or database modifications, a user replaces the original connector with a QueryBooster-provided connector. The connector forwards an original query from the application to the *Query Rewriter*, and sends the rewritten query to the database. When the database returns the result of the new query, the connector sends the query performance information back to the *Query Manager*. Note that the connector does not send the query result to QueryBooster. Thus it ensures high security as no user data goes through a third-party server.

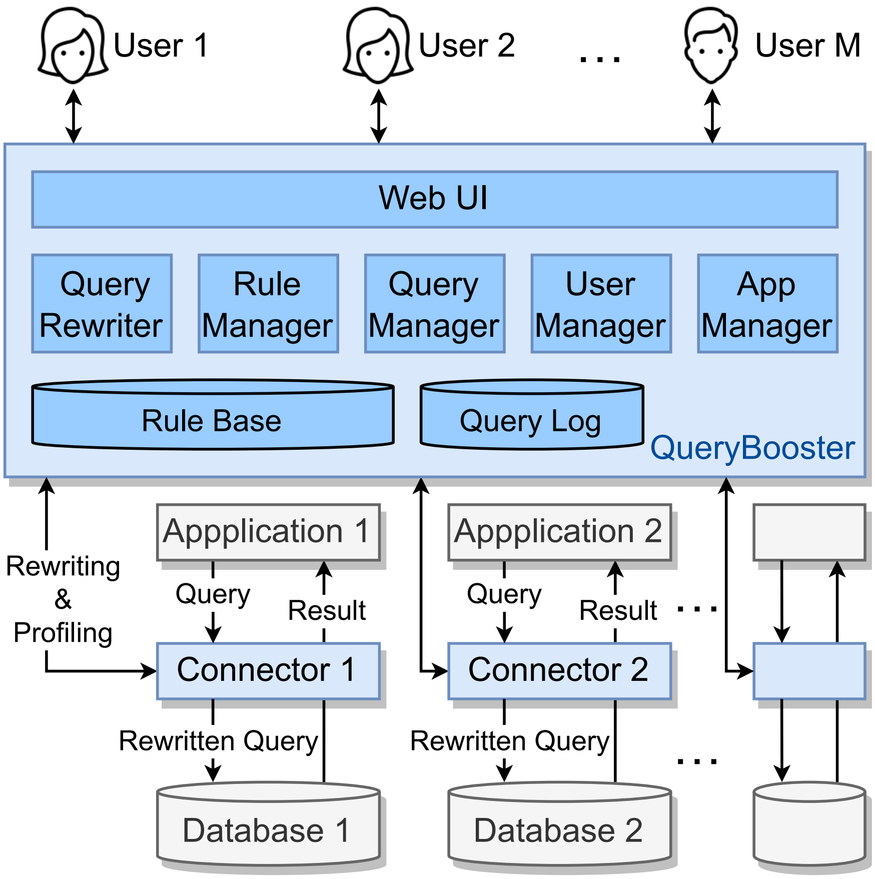


Figure 4.2: QueryBooster system architecture.

All the query rewriting path and corresponding performance information are stored in the *Query Log*. As a background process, the *Query Manager* periodically runs rewriting rules shared by different users against the workloads in the query log and marks those queries with suggested rules useful to the queries.

4.3 Demonstration of the User Experience

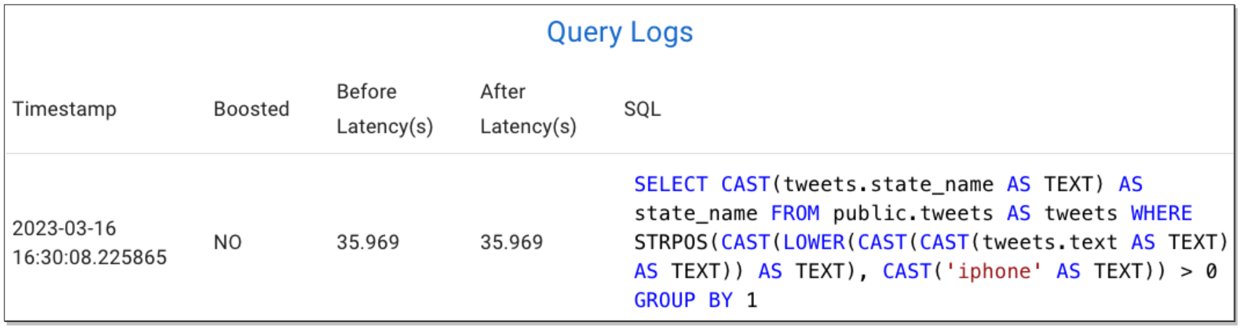
In this section, we use real-world applications and Twitter datasets to demonstrate the experience of the QueryBooster service for two users to rewrite their application queries and share rewriting knowledge.

Suppose Alice is a database administrator who manages a PostgreSQL database that stores tweet data to support a data analysis team. The team uses Tableau to study the spatial and temporal distributions of keyword-related tweets. After creating a Tableau dashboard on top of PostgreSQL, Alice tries a few visualization queries, and the performance is not satisfying. Therefore, she uses QueryBooster to rewrite the queries for better performance. Through QueryBooster's Web UI, she creates an application and receives an application GUID (Global Unique Identifier) generated from the system. Then, she downloads the provided JDBC driver and places it in the folder of Tableau connectors. Finally, she configures the JDBC driver with the provided application GUID. Alice has now completed setting up QueryBooster.

4.3.1 Formulating Rules through a Rule Language

To identify the queries with performance bottlenecks, Alice utilizes QueryBooster's query logging feature. She first tries the slow analytics operations on Tableau, and goes to the *Query Logs* page to check the SQL queries sent from Tableau to PostgreSQL. Figure 4.3

shows the Web UI illustrating the information of one query formulated by Tableau. It shows the query, its timestamp, latency, and whether it has been rewritten or not.



Query Logs

Timestamp	Boosted	Before Latency(s)	After Latency(s)	SQL

2023-03-16 16:30:08.225865	NO	35.969	35.969	SELECT CAST(tweets.state_name AS TEXT) AS
				state_name FROM public.tweets AS tweets WHERE
				STRPOS(CAST(LOWER(CAST(CAST(tweets.text AS TEXT)
				AS TEXT)) AS TEXT), CAST('iphone' AS TEXT)) > 0
				GROUP BY 1

Figure 4.3: The "Query Logs" page of QueryBooster shows information about queries from the application.

After investigating those queries, Alice identifies a lot of type-casting expressions (i.e., `CAST(··· AS TEXT)`), as shown in the query in Figure 4.3. Tableau adds those type-casting expressions to prevent computational overflow or datatype mismatching errors [116]. Based on her knowledge of the tweet data and its schema, Alice knows that the type-casting expressions are not necessary for the queries. Thus, Alice wants to input a rewriting rule to remove those type-castings from the queries. The VarSQL rule language is easy-to-use, and Alice manually formulates the rule (called "`Remove Cast Text`") as shown in Figure 4.4.

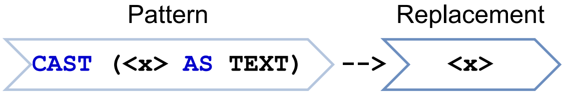
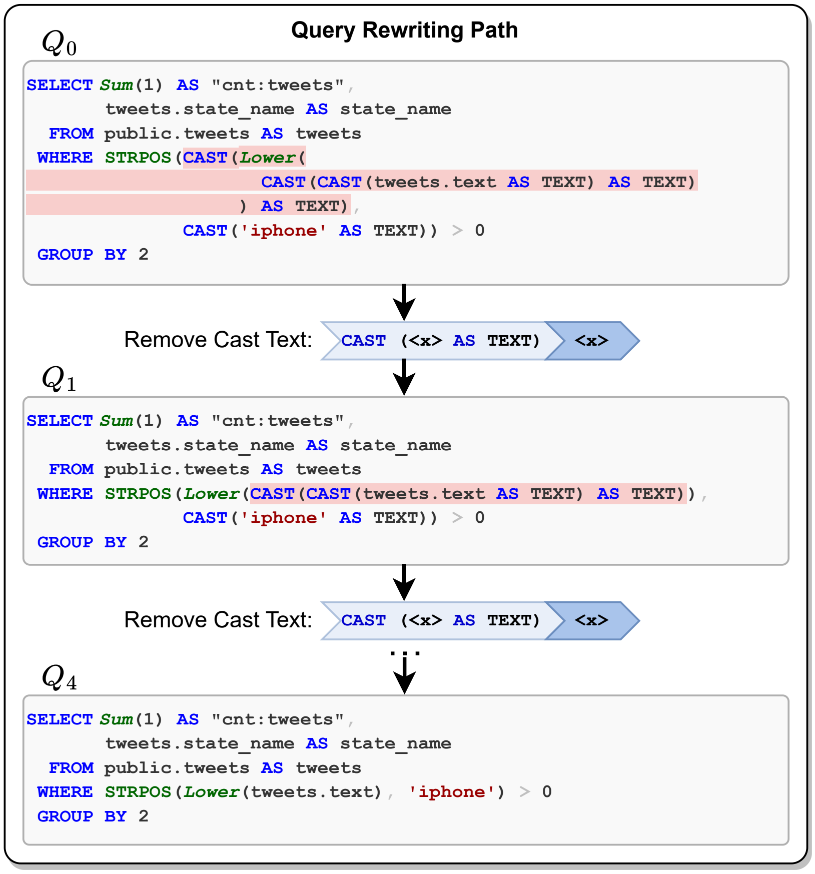


Figure 4.4: Alice formulates a rule to remove `CAST(··· AS TEXT)` expressions using the VarSQL language [88]. In VarSQL's syntax, `<x>` is an element-variable that represents a table, column, value, expression, predicate, or sub-query.

With the "`Remove Cast Text`" rewriting rule enabled, Alice tries the slow operations on Tableau again, but the performance is not improved. However, the query's rewriting path page (Figure 4.5) shows that the rule is correctly triggered, and the rewritten query is as expected. Therefore, Alice continues to optimize this query.



```
SELECT Sum(1) AS "cnt:tweets",
       tweets.state_name AS state_name
  FROM public.tweets AS tweets
 WHERE STRPOS(CAST(Lower(
                     CAST(CAST(tweets.text AS TEXT) AS TEXT)
                   ) AS TEXT),
              CAST('iphone' AS TEXT)) > 0
 GROUP BY 2
```

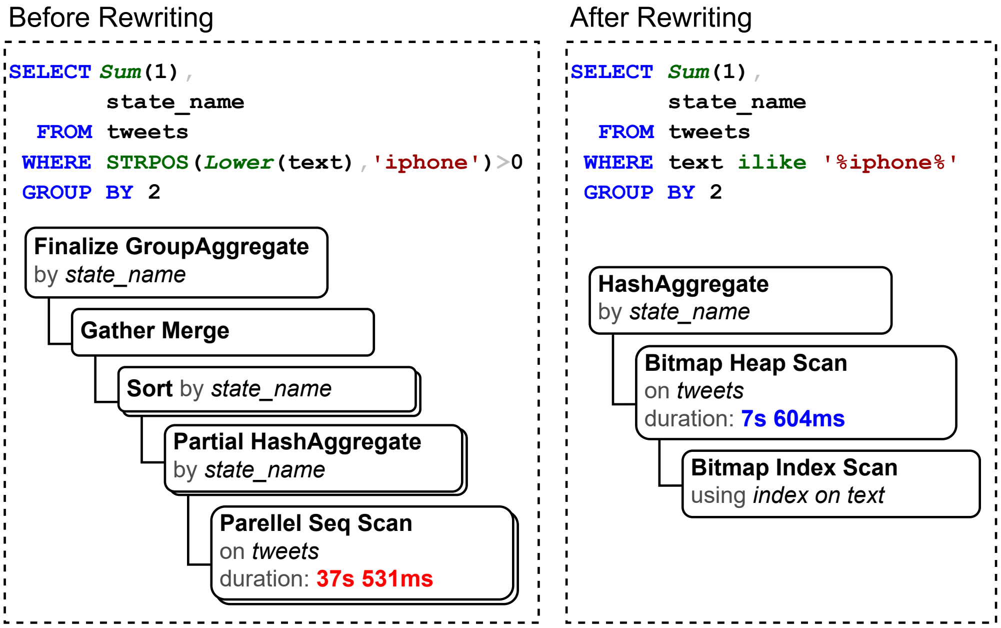
```
SELECT Sum(1) AS "cnt:tweets",
       tweets.state_name AS state_name
  FROM public.tweets AS tweets
 WHERE STRPOS(Lower(CAST(CAST(tweets.text AS TEXT) AS TEXT)),
              CAST('iphone' AS TEXT)) > 0
 GROUP BY 2
```

```
SELECT Sum(1) AS "cnt:tweets",
       tweets.state_name AS state_name
  FROM public.tweets AS tweets
 WHERE STRPOS(Lower(tweets.text), 'iphone') > 0
 GROUP BY 2
```

Figure 4.5: QueryBooster shows the rewriting path of a query.

4.3.2 Formulating Rules by Providing Examples

After a deeper investigation into the query and some online search, Alice finds that a trigram index on the ‘‘tweets’’.‘‘text’’ attribute in PostgreSQL supports wildcard filtering predicates such as LIKE and ILIKE. However, PostgreSQL fails to use this index because the wildcard predicate formulated by Tableau is STRPOS(LOWER(‘‘tweets’’.‘‘text’’), ’iphone’) > 0, which is equivalent to ‘‘tweets’’.‘‘text’’ ILIKE ’%iphone%’. Alice identifies that replacing the STRPOS()>0 predicate with the ILIKE predicate in the query can produce a much more efficient plan in PostgreSQL (as shown in Figure 4.6). Thus, Alice wants to introduce another rewriting rule that achieves this replacing logic. However, for this rewriting, Alice is not sure how to manually input the rewriting rule, so she uses the QueryBooster's rule-suggestion feature by providing a rewriting example.


Before Rewriting

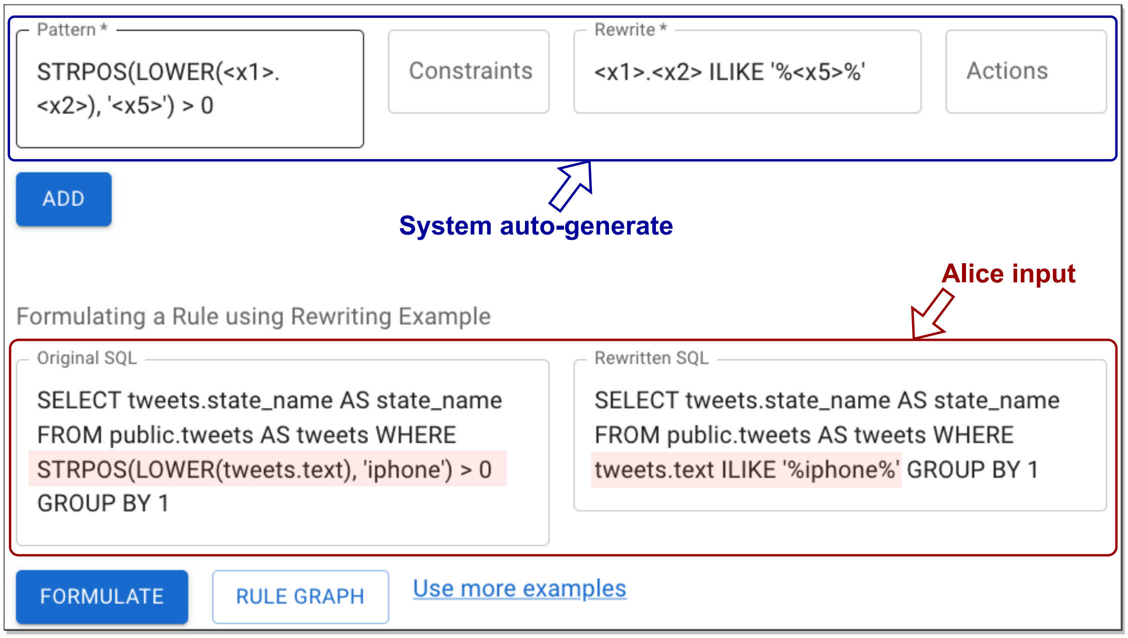
```
SELECT Sum(1) ,
       state_name
  FROM tweets
 WHERE STRPOS(Lower(text) , 'iphone')>0
 GROUP BY 2
```

After Rewriting

```
SELECT  Sum(1) ,
        state_name
  FROM  tweets
 WHERE  text ilike '%iphone%'
 GROUP  BY 2
```

Figure 4.6: Queries and plans before and after the rewriting of replacing the STRPOS()>0 predicate with the ILIKE predicate.

After copying and pasting the original query to both the original query and rewritten query text boxes on the *Rule Formulator* page, Alice modifies the rewritten query to her desired format and clicks the "Formulate" button. Next, QueryBooster automatically generalizes the example pair of queries into a rewriting rule that achieves the rewriting intention of Alice (as shown in Figure 4.7). Finally, Alice saves the new rewriting rule to the system with the name "`STRPOS To ILIKE`."



STRPOS(LOWER(<x1>.<x2>), '<x5>') > 0

<x1>.<x2> ILIKE '%<x5>%'

```
SELECT tweets.state_name AS state_name
FROM public.tweets AS tweets WHERE
STRPOS(LOWER(tweets.text), 'iphone') > 0
GROUP BY 1
```

```
SELECT tweets.state_name AS state_name
FROM public.tweets AS tweets WHERE
tweets.text ILIKE '%iphone%' GROUP BY 1
```

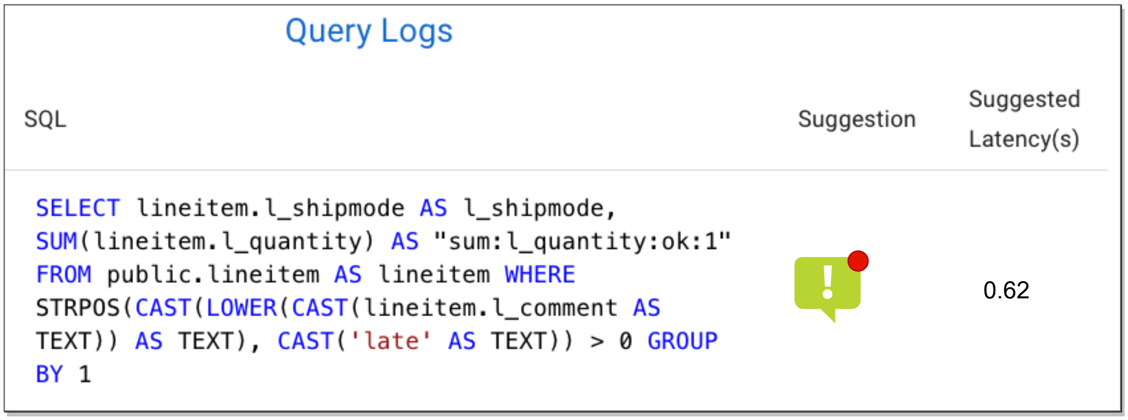
Figure 4.7: QueryBooster suggests a rule given a rewriting example.

With the two rewriting rules enabled, the query performance is significantly improved. For example, the original query takes 37.5 seconds, and the rewritten query only takes 7.6 seconds.

4.3.3 Suggesting Useful Rules from Other Users

Suppose Bob is a database administrator (DBA) and supports a business team who wants to analyze the TPC-H [114] dataset using Tableau on top of PostgreSQL. For one of the analytical queries in Bob's workload, QueryBooster identifies the potential of boosting the query performance by applying the two rewriting rules provided by Alice. QueryBooster automatically suggests the potential rewriting of the query to Bob, as shown in Figure 4.8.

Bob inspects the rewritten query through the "Rewriting Suggestion" page of QueryBooster



Query Logs

SQL	Suggestion	Suggested Latency(s)
SELECT lineitem.l_shipmode AS l_shipmode, SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:1" FROM public.lineitem AS lineitem WHERE STRPOS(CAST(LOWER(CAST(lineitem.l_comment AS TEXT)) AS TEXT), CAST('late' AS TEXT)) > 0 GROUP BY 1	!	0.62

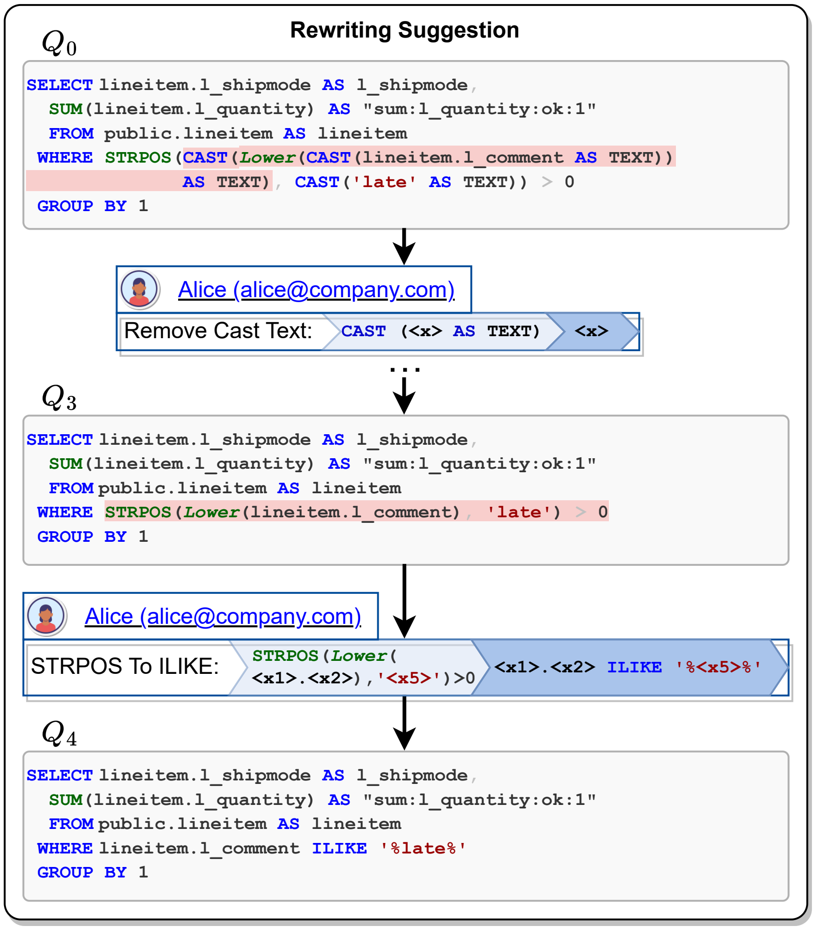
Figure 4.8: QueryBooster suggests Bob a rewriting for a query.

(Figure 4.9) and decides to enable these two rules for his application. Note that Bob may modify the rules slightly to fit his dataset, e.g., adding a "LOWER" function on top of the keyword constant to make sure the keyword constant is always lowercase. Bob then returns to Tableau and observes a significant query performance improvement.

To this end, we demonstrate the powerful experience of using the QueryBooster service to rewrite application queries and share knowledge.

4.4 Supporting Management and Sharing Rewritings

In this section, we demonstrate the experience of using QueryBooster to manage rewriting rules and discuss its ability to give users full control over the rewritings on their workloads. We then show how users share rewriting knowledge intuitively and interactively through the QueryBooster service and discuss the trade-off between privacy and accuracy of using different approaches to suggesting rewritings.



```
SELECT lineitem.l_shipmode AS l_shipmode,
  SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:1"
  FROM public.lineitem AS lineitem
 WHERE STRPOS(CAST(Lower(CAST(lineitem.l_comment AS TEXT))
              AS TEXT), CAST('late' AS TEXT)) > 0
 GROUP BY 1
```

```
CAST (<x> AS TEXT)  →  <x>
```

```
SELECT lineitem.l_shipmode AS l_shipmode,
  SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:1"
  FROM public.lineitem AS lineitem
 WHERE STRPOS(Lower(lineitem.l_comment), 'late') > 0
 GROUP BY 1
```

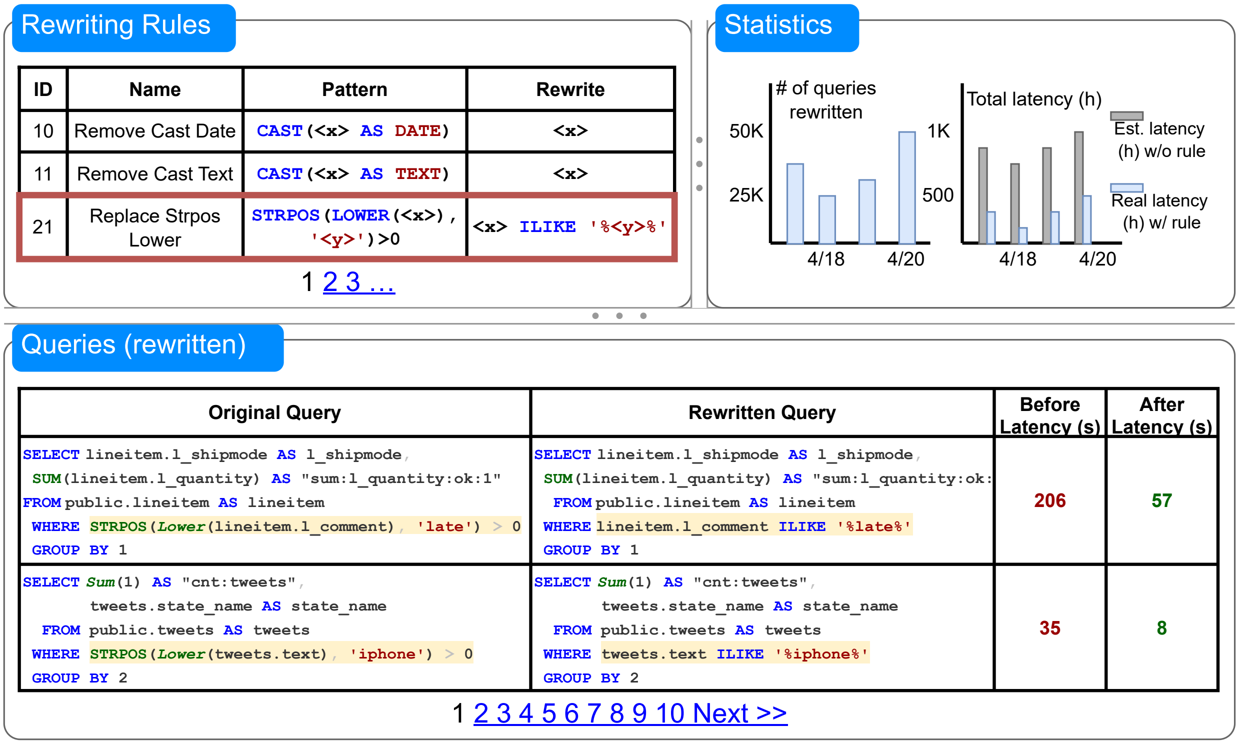
```
STRPOS(Lower(
<x1>.<x2>),'<x5>')>0  →  <x1>.<x2> ILIKE '%<x5>%'
```

```
SELECT lineitem.l_shipmode AS l_shipmode,
  SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:1"
  FROM public.lineitem AS lineitem
 WHERE lineitem.l_comment ILIKE '%late%'
 GROUP BY 1
```

Figure 4.9: QueryBooster suggests a rewriting for Bob's query by applying Alice's two rules.

4.4.1 Informative and Fully-Controllable Rewriting Management

In this section, we show that QueryBooster provides a user experience to manage rewriting rules and gives the users full control over the rewritings on their query workloads.



Rewriting Rules

ID	Name	Pattern	Rewrite
10	Remove Cast Date	`CAST(<x> AS DATE)`	`<x>`
11	Remove Cast Text	`CAST(<x> AS TEXT)`	`<x>`
21	Replace Strpos Lower	`STRPOS(LOWER(<x>), '<y>')>0`	`<x> ILIKE '%<y>%'`

1 2 3 ...

Statistics

Queries (rewritten)

Original Query	Rewritten Query	Before Latency (s)	After Latency (s)

`SELECT lineitem.l_shipmode AS l_shipmode,`
`SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:1"`
`FROM public.lineitem AS lineitem`
`WHERE STRPOS(Lower(lineitem.l_comment), 'late') > 0`
`GROUP BY 1`
	`SELECT lineitem.l_shipmode AS l_shipmode,`
	`SUM(lineitem.l_quantity) AS "sum:l_quantity:ok:`
	`FROM public.lineitem AS lineitem`
	`WHERE lineitem.l_comment ILIKE '%late%'`
	`GROUP BY 1`
		206	57

`SELECT Sum(1) AS "cnt:tweets",`
`tweets.state_name AS state_name`
`FROM public.tweets AS tweets`
`WHERE STRPOS(Lower(tweets.text), 'iphone') > 0`
`GROUP BY 2`
	`SELECT Sum(1) AS "cnt:tweets",`
	`tweets.state_name AS state_name`
	`FROM public.tweets AS tweets`
	`WHERE tweets.text ILIKE '%iphone%'`
	`GROUP BY 2`
		35	8

1 2 3 4 5 6 7 8 9 10 Next >>

Figure 4.10: The dashboard of rewriting-rule analysis in QueryBooster.

Showing rewriting effect and performance statistics of rewriting rules. In the early discussions, we presented a user experience of using QueryBooster to analyze queries and the rewriting effects on them. Now we show a user experience of using QueryBooster to analyze rewriting rules and their effects on query workloads. Figure 4.10 shows the dashboard of a rewriting rule chosen by a user. The dashboard shows the statistics of how the rewriting rule affects the query workloads, such as how many queries have been rewritten using this rule, how much query execution time has been saved by using this rule to rewrite queries, etc. In addition, the dashboard includes the rewriting history of using the specific rule to rewrite queries, such as its original and rewritten queries. Using this informative dashboard, users can gain better insights into the effectiveness of their rewriting rules and make wise decisions about whether to apply the same set of rules on broader applications or refine some rewriting rules to avoid performance regression of the workloads.

To support such a dashboard, we design an ER diagram for the database of QueryBooster, as shown in Figure 4.11.

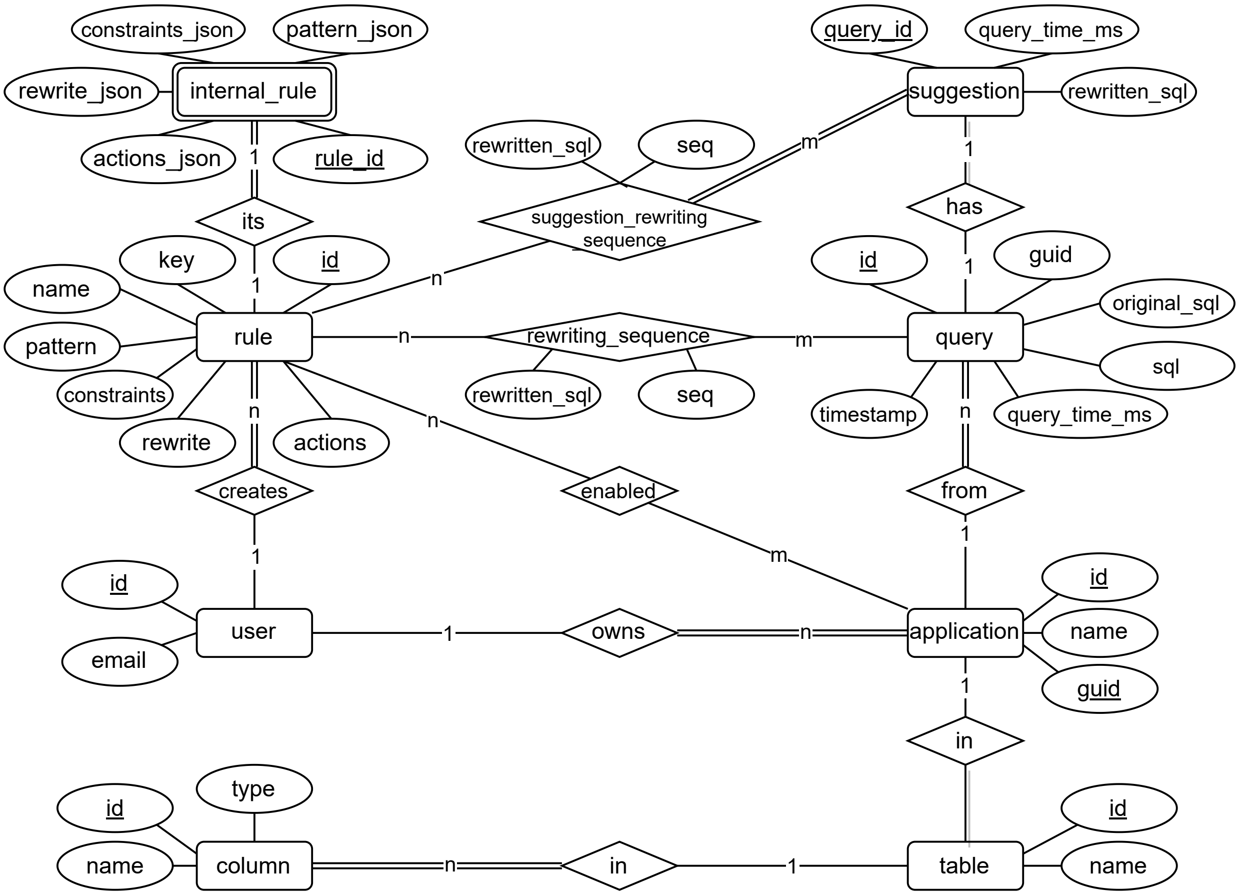
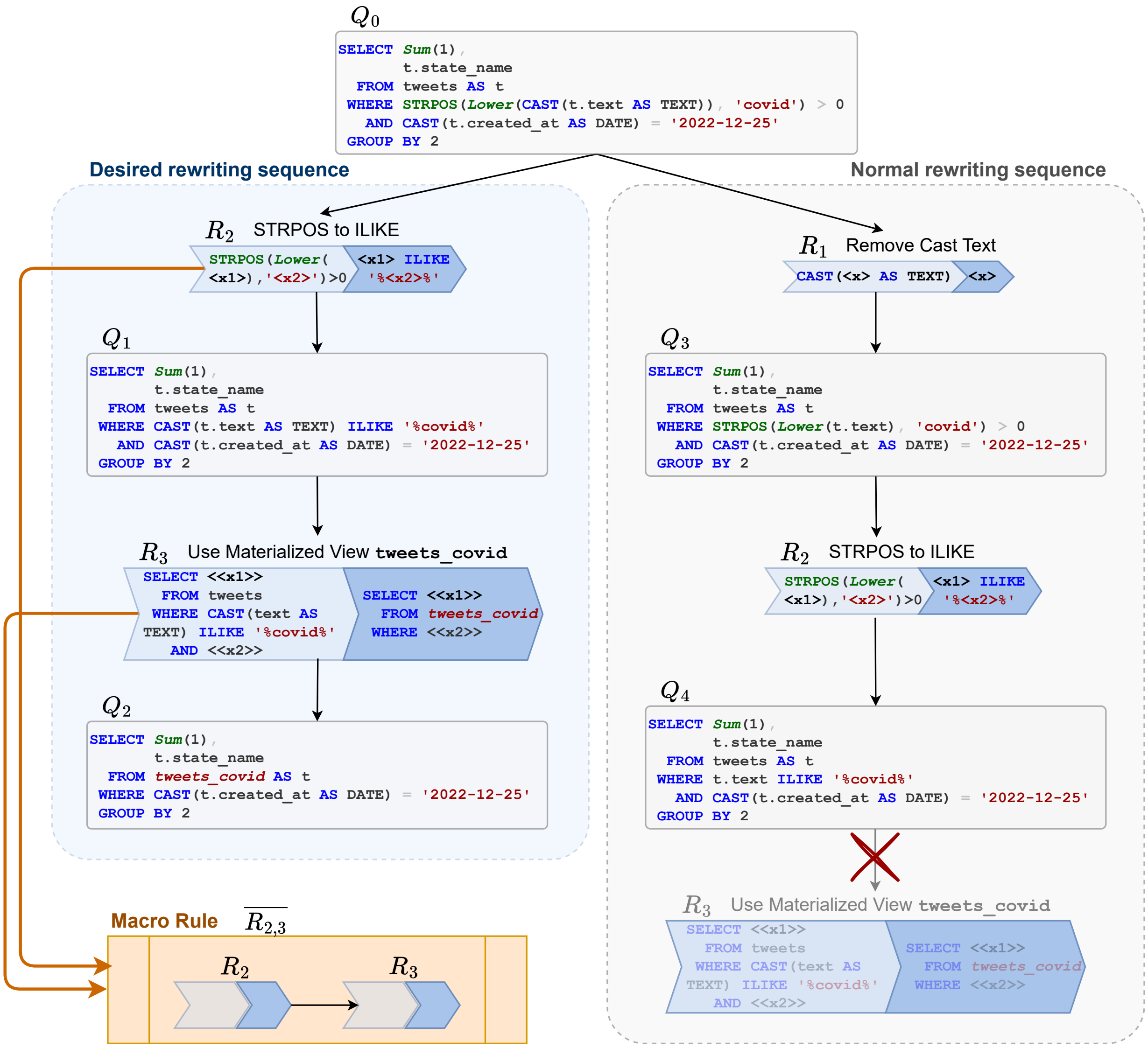


Figure 4.11: The ER diagram of the database of QueryBooster.

We keep track of a sequence of rewriting an original query using different rewriting rules in the `rewriting_sequence` relation. Each tuple in `rewriting_sequence` is one rewriting step. The `sequence` numbers indicate the order of the steps. Each step includes the `rule_id` of the rewriting rule applied and the `rewritten_sql` after applying the rule. The rewriting sequences of each original query can be obtained internally when the rewriting engine rewrites a query. The query latency is stored in the `query` relation. If the QueryBooster service cannot have access to the users' databases, we track the query latency through the database connector (e.g., JDBC/ODBC drivers) customized by QueryBooster. This approach is non-intrusive compared to other approaches and provides high security and privacy for the QueryBooster service. We will further discuss the details in Section 4.5.


Q_0
Desired rewriting sequence
Normal rewriting sequence
R_2 STRPOS to ILIKE
Q_1
R_3 Use Materialized View tweets_covid
Q_2
R_1 Remove Cast Text
Q_3
R_2 STRPOS to ILIKE
Q_4
R_3 Use Materialized View tweets_covid
Macro Rule $\overline{R_{2,3}}$
R_2
R_3

```
SELECT Sum(1),
       t.state_name
  FROM tweets AS t
 WHERE STRPOS(Lower(CAST(t.text AS TEXT)), 'covid') > 0
   AND CAST(t.created_at AS DATE) = '2022-12-25'
 GROUP BY 2
```

```
STRPOS(Lower(<x1>),'<x2>')>0    <x1> ILIKE '%<x2>%'
```

```
SELECT Sum(1),
       t.state_name
  FROM tweets AS t
 WHERE CAST(t.text AS TEXT) ILIKE '%covid%'
   AND CAST(t.created_at AS DATE) = '2022-12-25'
 GROUP BY 2
```

```
SELECT <<x1>>                         SELECT <<x1>>
  FROM tweets                           FROM tweets_covid
 WHERE CAST(text AS                    WHERE <<x2>>
TEXT) ILIKE '%covid%'
   AND <<x2>>
```

```
SELECT Sum(1),
       t.state_name
  FROM tweets_covid AS t
 WHERE CAST(t.created_at AS DATE) = '2022-12-25'
 GROUP BY 2
```

```
CAST(<x> AS TEXT)    <x>
```

```
SELECT Sum(1),
       t.state_name
  FROM tweets AS t
 WHERE STRPOS(Lower(t.text), 'covid') > 0
   AND CAST(t.created_at AS DATE) = '2022-12-25'
 GROUP BY 2
```

```
STRPOS(Lower(<x1>),'<x2>')>0    <x1> ILIKE '%<x2>%'
```

```
SELECT Sum(1),
       t.state_name
  FROM tweets AS t
 WHERE t.text ILIKE '%covid%'
   AND CAST(t.created_at AS DATE) = '2022-12-25'
 GROUP BY 2
```

```
SELECT <<x1>>                         SELECT <<x1>>
  FROM tweets                           FROM tweets_covid
 WHERE CAST(text AS                    WHERE <<x2>>
TEXT) ILIKE '%covid%'
   AND <<x2>>
```

Figure 4.12: An example of using macro rules to enforce a user-desired rewriting sequence for a query.

Using macro rules to allow users to control the rewriting sequences. As mentioned earlier, different orders to apply the same set of rules on a query may result in different rewritten queries. QueryBooster provides a mechanism for users to specify the priority of each rule, such that the users can enforce a partial order of applying the same set of rules for a query. However, there are cases where for different queries, the users may want different orders of applying the rules for particular rewriting sequences that can improve their query performance. Figure 4.12 shows an example query with two different rewriting sequences if applying the same set of rules in two different orders. The typical rewriting sequence shown on the right-hand side fails to apply rule R_3 to utilize the materialized view `tweets_covid` to accelerate the query. The user prefers the rewriting sequence on the left-hand side, which applies rule R_2 before rule R_1 (assuming the priority of R_1 is higher than R_2) and produces query Q_1 that can match the pattern of R_3, which can further rewrite the query to a more efficient form.

To solve this problem and give the users full control over the rewritings of their query workloads, QueryBooster provides a rewriting-rule management framework where users can bundle a sequence of rules as a "**macro-rule**". A macro-rule has the highest priority and will be checked against a given query before applying any basic rules. QueryBooster also checks the pattern of a new input macro-rule against existing macro-rules to enforce that no two macro-rules can match the same query. Then, any query that can match the sequence of rules in a macro-rule will be enforced into the rewriting sequence desired by the user specified in the macro-rule. Note that using a macro-rule to rewrite a query is an atomic process, which means that each intermediate rewritten query inside a macro-rule has to match the pattern of the following rule in the sequence. If any intermediate step fails in the pattern-matching, the entire macro-rule is treated as not matching the original query's pattern.

4.4.2 Intuitive and Interactive Sharing of Rewriting Knowledge

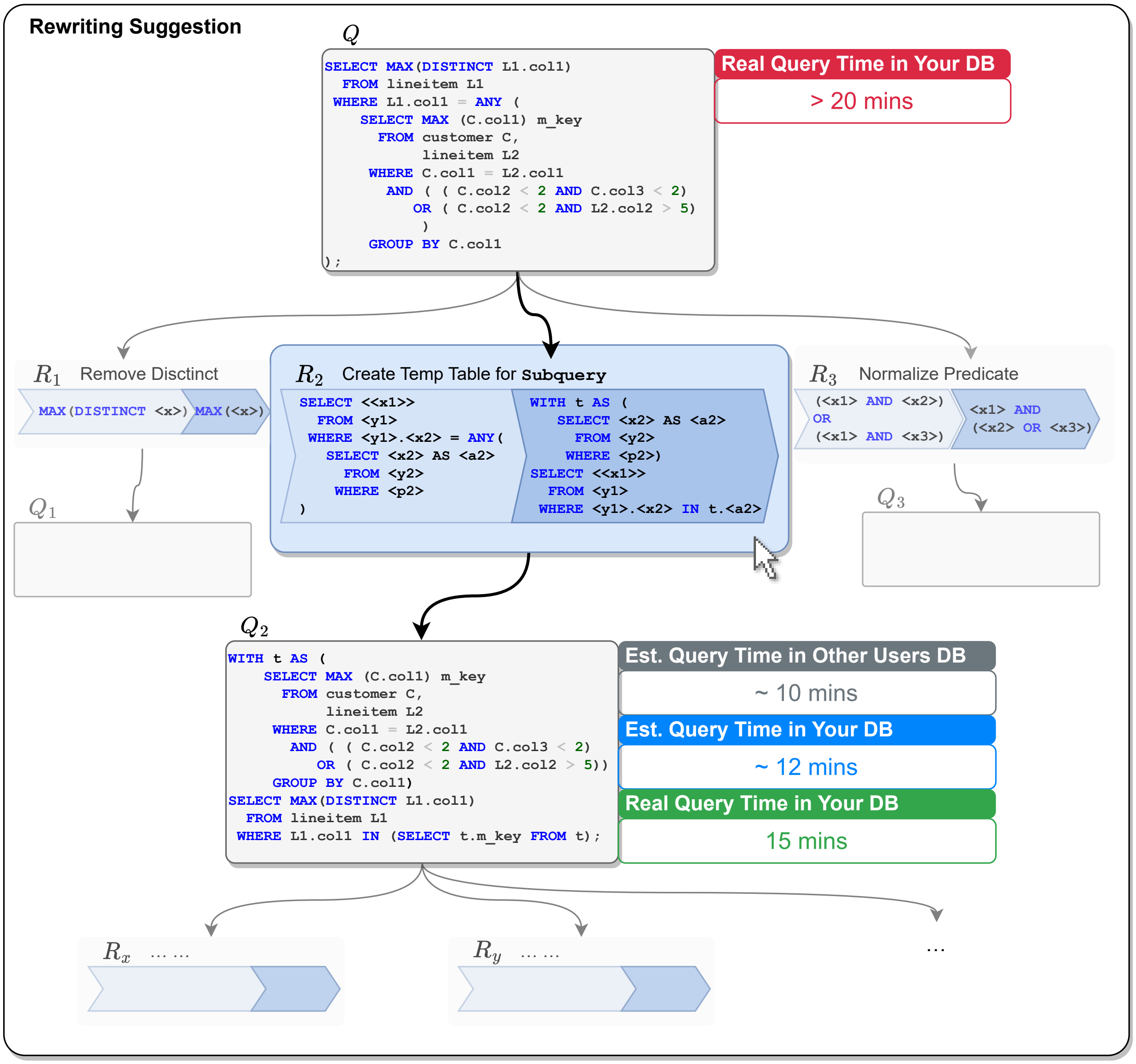
In this section, we first discuss a policy of sharing rewriting rules among users. Then, we show the user experience that QueryBooster can suggest different rewriting sequences to users interactively. Finally, we discuss the trade-off between privacy and accuracy of using different methods to estimate the benefits of different rewriting sequences.

A policy of sharing rewriting rules. One major advantage of using a centralized service of QueryBooster to rewrite queries is allowing users to share their rewriting knowledge. Rewriting rules accumulated from different users and applications can be an invaluable asset to the entire user community. However, a major concern is maintaining the privacy and security of users' data. In QueryBooster, we propose the following policy to keep a high privacy and security level for users' data. First, the queries should always be protected and not revealed to third parties. The user account used to download the database connector and collect those queries is the owner of the collected queries. Second, we only allow sharing of rewriting rules. We also do not allow sharing of rewriting rules with sensitive information. For example, if the pattern of a rewriting rule is a SQL query containing specific table or column names, we can disallow the rewriting rule to be shared. In this case, both the user queries and database schema are protected. Third, we adopt a common sharing model as the policy of sharing rewriting rules among different users. A user can only retrieve rewriting suggestions using other users' rewriting rules if the user agrees to share their own rewriting rule base. In this sharing model, everyone contributes and benefits.

Suggesting different rewriting sequences interactively. Users can analyze their query performance through QueryBooster, which also suggests rewriting rules shared by other users for their slow queries. As shown in Figure 4.13, when a user selects a slow query in her application workload, a rewriting-suggestion dashboard will show potential rewriting rules applicable to the query. As discussed earlier, for a particular query and a set of applicable rewriting rules, there can be different orders to rewrite the query, which may result in different final forms of the rewritten query. To allow users to control their rewriting sequences, QueryBooster also shows different rewriting rules to the user interactively, where the user can select which rule to apply first. For example, in Figure 4.13, for the slow query Q, the user first clicks the rule R_2 (i.e., "Create Temp Table for Subquery") to rewrite it and sees the rewritten query Q_2. For the new rewritten query Q_2, QueryBooster shows a new set of applicable rules for the user to choose and repeats this process. Another important piece of information for users to decide whether to adopt the suggested rules is the query performance improvement of different rules or rewriting sequences. QueryBooster provides this information to the users in the process of suggesting rewriting rules. For example, QueryBooster shows the real query time of running the original query Q in the user's database, as shown in the red box. For the new rewritten query Q_2, it also shows the estimated query time or real query time obtained from different sources based on user selection or the system configuration. For example, based on the information that the rewriting rule R_2 accelerates queries in other users' workloads, QueryBooster can show a rough estimation of how much time can be saved if the rule is applied to the query Q in this workload, as shown in the grey box. Similarly, if given access to the target database of the query Q, QueryBooster can use the target database's optimizer to better estimate the query time of the rewritten query Q_2, as shown in the blue box. However, different estimations may require different security levels and also use different system resources. Thus, we let the users choose whether to conduct the estimation or even run the query for the benefit of rewriting the query. In this way, QueryBooster enables the users to manage their rewritings intuitively and interactively.

Different methods to estimate the benefits of rewriting sequences. When suggesting rules for users' queries, an intuitive metric for users to understand the benefit of rewriting a query using a particular rule is the query time acceleration, i.e., how much running time can be saved if we rewrite the original query. In practice, it is not straightforward to obtain this information, given the restrictions of access level security and the amount of resources needed. In QueryBooster, we consider three different methods to estimate the benefit of improving the running time of queries for a rule suggestion.

The first method is to estimate the query performance improvement based on the reduction ratio of the running time of queries in other users' workloads. For example, suppose the rule


Rewriting Suggestion
Q
Real Query Time in Your DB
> 20 mins
R_1 Remove Disctinct
Q_1
R_2 Create Temp Table for Subquery
R_3 Normalize Predicate
Q_3
Q_2
Est. Query Time in Other Users DB
~ 10 mins
Est. Query Time in Your DB
~ 12 mins
Real Query Time in Your DB
15 mins
R_x … …
R_y … …
…

```
SELECT MAX(DISTINCT L1.col1)
  FROM lineitem L1
 WHERE L1.col1 = ANY (
    SELECT MAX (C.col1) m_key
      FROM customer C,
           lineitem L2
     WHERE C.col1 = L2.col1
       AND ( ( C.col2 < 2 AND C.col3 < 2)
          OR ( C.col2 < 2 AND L2.col2 > 5)
           )
     GROUP BY C.col1
);
```

```
MAX(DISTINCT <x>)
```

```
MAX(<x>)
```

```
SELECT <<x1>>
  FROM <y1>
 WHERE <y1>.<x2> = ANY(
   SELECT <x2> AS <a2>
     FROM <y2>
    WHERE <p2>
)
```

```
WITH t AS (
   SELECT <x2> AS <a2>
     FROM <y2>
    WHERE <p2>)
SELECT <<x1>>
  FROM <y1>
 WHERE <y1>.<x2> IN t.<a2>
```

```
(<x1> AND <x2>)
OR
(<x1> AND <x3>)
```

```
<x1> AND
(<x2> OR <x3>)
```

```
WITH t AS (
    SELECT MAX (C.col1) m_key
      FROM customer C,
           lineitem L2
     WHERE C.col1 = L2.col1
       AND ( ( C.col2 < 2 AND C.col3 < 2)
          OR ( C.col2 < 2 AND L2.col2 > 5))
     GROUP BY C.col1)
SELECT MAX(DISTINCT L1.col1)
  FROM lineitem L1
 WHERE L1.col1 IN (SELECT t.m_key FROM t);
```

Figure 4.13: QueryBooster suggests rewriting rules interactively. The user can select interesting rules and see the corresponding rewritten query by applying the rule and the estimated or real query time of both the original and rewritten queries. Users can explore different rewriting rules and sequences based on their database and rewriting needs.

R_2 in Figure 4.13 was formulated by a user (Alice) and has been used on her workload for a while. QueryBooster accumulates the query running time of both the original and rewritten queries of rule R_2 in Alice's workload. Therefore, QueryBooster can compute an aggregated ratio of how much the query time is reduced by utilizing the R_2 to rewrite queries in Alice's workload. Then for a new query Q analyzed by a different user (called Bob), we can compute the estimated query time of the rewritten query Q_2 after applying R_2 on Q based on the ratio and the original query Q's query time. The main advantage of this method is that it does not require any additional access to Bob's database, and the computation is fast due to its low cost. However, the disadvantage of this method is its low accuracy. In most cases, a query's running time depends on the database software and the dataset's properties, such as its distribution, volume, and available indexes. The accuracy of the estimation can be low since it is based on other users' datasets.

The second method is to use the target database's query optimizer to estimate the running time of rewritten queries. For example, suppose the user Bob configures QueryBooster with the username and password of his target database and gives QueryBooster access to use the "`EXPLAIN`" command against its query optimizer. QueryBooster can send an `EXPLAIN` statement for each of the rewritten queries generated by applying the suggested rewriting rules. Usually, the `EXPLAIN` command only shows an estimated cost of a query instead of its running time. In this case, we can `EXPLAIN` the original query to get its estimated query cost and compare the costs of the original query and the rewritten query as the benefit of the rewriting rule. Compared to the first method, this method gives a much more accurate estimation since it uses the target database's optimizer, which depends on the target dataset. However, the disadvantage is that it requires access to the target database and a high cost of explaining the query plan for each rewritten query.

The third method is to directly run the rewritten query against the target database to obtain its real running time. Apparently, this method requires Bob to give QueryBooster full access to the target database to run queries against his business dataset. The advantage is we can obtain the accurate benefit of applying a suggested rewriting rule on an original query since the running time of both the original and the rewritten queries are accurate. The disadvantages include the highest level of access to the user's database and running the rewritten queries, which introduces a high cost of using resources of the user's database (e.g., computation, memory, etc.).

In summary, different methods require different levels of security access and different resources to compute. In return, they offer different levels of accuracy. We can provide all of them and allow users to choose appropriate methods based on their needs.

4.5 Minimizing Database Intrusiveness

In this section, we first study the intrusiveness levels of different design choices to develop a SQL query rewriting system and their trade-offs. We then discuss the unique problems encountered in developing our proposed SQL query rewriting service and solutions. Particularly, we first survey the connector licenses of common database vendors to show the applicability of our proposed QueryBooster service. We then discuss how to customize the database connectors to profile application queries' performance.

4.5.1 Intrusiveness of SQL Query Rewriting Systems

In this section, we study the architectures of three typical SQL query rewriting systems on the market (including the proposed QueryBooster system) and discuss the intrusiveness of their different design choices. Table 4.1 summarizes the three systems.

EverSQL [27] is a cloud-based software tool designed to help developers and database administrators (DBAs) optimize the performance of their SQL queries. Users copy their application SQL queries and paste them to the Web-based input box provided by EverSQL. EverSQL then analyzes the queries and provides optimization suggestions based on known database performance patterns and AI techniques. By highlighting problematic SQL snippets and pointing out missing indexes, EverSQL aims to assist developers and DBAs in optimizing their queries to achieve better performance. Without any integration into the users' application and database layer, EverSQL is non-intrusive, for which we name the intrusiveness level *L0*. The advantage of this *L0* intrusiveness is that EverSQL has good security and privacy, as no user queries or data go through a third-party server. (Although EverSQL still sees the query string copied by the user, the user can easily "sanitize" the query string by removing sensitive information before pasting it to the EverSQL.) However, the disadvantage is that it cannot rewrite online queries automatically and requires developers to modify the application code to apply the optimized rewriting.

Table 4.1: Intrusiveness levels of systems.

System	**EverSQL**	**Squidster**	**Keebo**
Intrusiveness Level	L0	L1	L2
Approach	copy & paste query strings	customize DB connectors to rewrite SQL strings	manipulate SQL strings and data through a middleware server
Automatic Rewrite Online Queries	No	Yes	Yes
Query through a Third-Party Server	No	Yes	Yes
Data through a Third-Party Server	No	No	Yes

QueryBooster provides a powerful and easy-to-use rule language (VarSQL) for users to define rewriting rules and also suggests rewriting rules to users. The QueryBooster service also provides features that allow users to profile their application workloads, manage their rewriting rules, and share rewriting knowledge with the community. QueryBooster minimizes its intrusiveness into the SQL query lifecycle of user applications and databases. Users do not need to modify any code of the application or database. The only change on the user side is to replace the original DB connector (e.g., a JDBC/ODBC driver) with a slightly customized version provided by QueryBooster. In this architecture, no data goes through a third-party server. We name the intrusiveness level *L1*. The advantages of this level are that QueryBooster does not introduce much overhead of transferring data between the application and the database, and QueryBooster provides a high privacy and security level without seeing any user data. In addition, QueryBooster does not require users to grant access to the target database.

Keebo [47] is a middleware system sitting between the application and the database layers. It uses machine learning and approximate query processing (AQP) techniques to accelerate analytical queries by rewriting online queries automatically to use offline-built data models or materialized views. Keebo sits on top of existing databases and provides native interfaces of them to applications. To use Keebo, users need to modify the entry point configuration of applications to point it to the Keebo server instead of the original database server. Once Keebo is installed, in its offline analysis phase, it will run a series of analytical queries and data manipulation commands to the user database to learn models and create materialized views. Then, for online application queries, Keebo will rewrite the queries to use those learned data models and materialized views to improve the query performance and assemble the result dataset. In this architecture, both the SQL queries and database are exposed to the Keebo's server. We name the intrusiveness level *L2*. The advantage of the *L2* intrusiveness is that the Keebo service has full control of how to rewrite queries and assemble the result. The disadvantages are two folds. The first is the overhead of occupying the target database resources to build data models and create materialized views and online query result data going through a third-party server. The second is the low privacy and security level in which a third-party server sees all user data and query results.

In summary, the proposed QueryBooster service benefits from automatically rewriting online application queries to improve performance by sacrificing a minimum level of intrusiveness in the users' application and database stack. Providing a customized database connector, QueryBooster requires no code change to the application or database. It provides high privacy and security without user data going through a third-party server. Providing a powerful and easy-to-use interface for users to manage their rewritings on their application queries, QueryBooster helps developers and DBAs improve their application performance efficiently.

4.5.2 A Survey of Database Connector Licences

QueryBooster provides customized database *Connectors* for users to download. A user replaces the original connector without application or database modifications with a QueryBooster-provided connector. The connector forwards an original query from the application to QueryBooster's server and sends the rewritten query to the database. When the database returns the query result, the connector sends the query performance information back to QueryBooster. Note that the connector does not send query results to QueryBooster.

In QueryBooster's architecture, the customized database connector is a critical module. Therefore, we surveyed popular database vendors for the licenses of their JDBC/ODBC connectors to validate the applicability of QueryBooster to these databases. The result is shown in Table 4.2.

Table 4.2: JDBC driver licenses for database vendors.

Database	**License**	**Open source?**	**Redistribution?**
MS SQL Server	MIT	Yes	Yes
MySQL	GNU GPL V2	Yes	Yes
Oracle	Oracle Free	Yes	*No*
PostgreSQL	BSD-2-Clause	Yes	Yes
Snowflake	Apache-2.0	Yes	Yes

Most database vendors provide JDBC/ODBC drivers with an open-source license. For such drivers, QueryBooster provides a slightly modified version of the driver that communicates with the proposed QueryBooster service to rewrite queries for these databases. For instance, we added only 112 lines of code to the PostgreSQL JDBC driver to develop the customized version. For databases with redistribution restrictions on their drivers (e.g., Oracle JDBC driver [73]), QueryBooster provides a software patch for users to compile the customized driver themselves. For applications and databases that communicate through a RESTful interface, QueryBooster provides a proxy web server that intercepts requests between them transparently. We assume the RESTful API endpoint in the application is configurable, i.e., we can switch the target database endpoint to our service.

4.5.3 Profiling Application Queries through Database Connectors

In earlier discussions, we show that through QueryBooster, users can analyze the performance of historical queries and the effects of rewriting rules on improving them. In addition, QueryBooster also supports showing aggregated performance of rewriting rules on application workloads. To support these features, QueryBooster needs to profile both the original application queries and their rewritten queries about their running time against the backend database.

There are multiple ways to fulfill this need. Many database performance monitoring systems [8, 111] profile workloads by looking at the native logs of the databases. This approach's main advantage is its non-intrusiveness; it does not run any third-party code on top of the target databases. However, to use this approach, database administrators must reconfigure the database to enable more informative logging modes to expose performance information and historical query statements in database logs. This reconfiguration usually requires a high level of privilege and may significantly increase the volume of logs. Another disadvantage of this approach is that the logging information might be obsolete as the dataset evolves, the software updates, and the configurations change. Another approach is running the queries

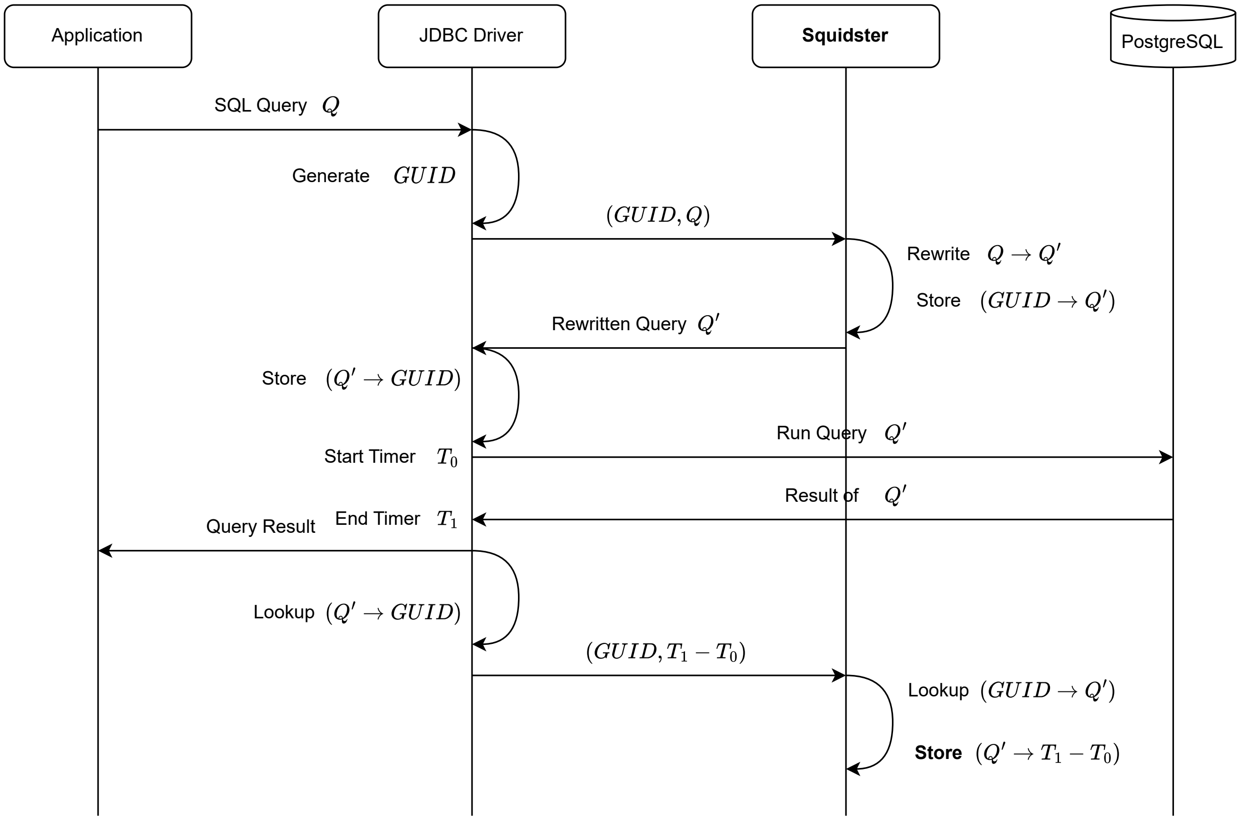


Figure 4.14: The workflow of profiling the database running time of a rewritten query generated by the QueryBooster rewriting service through a PostgreSQL JDBC driver.

against the target databases when we need the queries' performance profiles. The main advantage of this approach is that it has the most recent information. However, this approach is very intrusive to the target databases. It requires users to grant the system full access to run queries against the target databases and can occupy a significant amount of resources of the target database.

In QueryBooster, we propose a new approach that combines the advantages of both approaches mentioned above to minimize intrusiveness and maintain the freshness of the profiling result. Since we already provide a customized database connector to replace the original database connectors between the applications and the databases, we utilize the opportunity of opening the logic of the connectors to add query performance profiling logic into it. The main idea is to profile the original and rewritten queries' performance during the online query rewriting phase and store the profiling result back to the QueryBooster server.

Consider the JDBC driver of PostgreSQL as an example. Figure 4.14 shows the workflow to profile the database running time of a rewritten query generated by QueryBooster. When the application sends a SQL query Q to the JDBC driver, the driver first generates a GUID (globally unique identifier) for Q. Next, the driver sends the original query Q along with its GUID to the QueryBooster's online rewriting service to rewrite the query into Q'. QueryBooster stores the mapping between the GUID and the rewritten query Q' in its internal storage and then sends the rewritten query Q' back to the JDBC driver. The driver also stores the mapping between the query Q' and its GUID into an in-memory hash table, and then executes the query Q' against the PostgreSQL database. This execution is a synchronous function call; thus, the driver can profile Q''s running time by starting a timer before calling the function and ending the timer after the function returns. Once the query result is returned from the database to the driver, the driver first forwards the result back to the application without any delay, then looks up the query Q''s corresponding GUID from the in-memory hash table, and finally sends the profiling result along with the GUID back to QueryBooster. QueryBooster uses the GUID to look up its corresponding query string Q', and then stores the running time of Q' as the profiling result in its internal store. In this way, all queries going through the JDBC driver will be profiled and stored in QueryBooster's internal store to support its query analysis and rewriting management services.

4.6 Conclusions

In this chapter, we detailed the design and architecture of QueryBooster, a middleware-based multi-user system to provide query rewriting between applications and databases as a service. First, we used real-world applications and datasets to demonstrate the user experience of using QueryBooster to analyze application workloads and introduce new rewriting rules to improve the query performance. We also demonstrated the experience of using QueryBooster to manage rewriting rules and showed its ability to allow users to control the rewriting sequences of applying different rules on their queries. Second, we showed how users share rewriting knowledge through the QueryBooster service and discussed the trade-off between privacy and accuracy of using different approaches to suggest rewritings. Finally, we studied the intrusiveness levels of different design choices to develop a SQL query rewriting system and their trade-offs. We discussed the unique problems encountered in developing our proposed SQL query rewriting service and their solutions. Particularly, we surveyed the connector licenses of common database vendors to show the applicability of our proposed QueryBooster service. We also presented how to customize the database connectors to profile application queries' performance in QueryBooster.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

In this thesis, we presented middleware-based solutions to help databases optimize SQL queries through rewriting.

In Chapter 2, we studied how to rewrite database queries to improve execution performance in middleware-based visualization systems. We explored two optimization options, namely adding hints and making approximations. We developed a novel solution called Maliva, which adopts a Markov Decision Process (MDP) model to rewrite a visualization request under a tight time constraint. We gave a full specification of the solution, including how to construct an MDP model, train an agent, and use approximating rewriting options. Our experiments on both real and synthetic datasets showed that Maliva performed significantly better than the baseline without no-rewriting options in terms of both the probability of serving a visualization request within a time budget and query execution time.

In Chapter 3 and Chapter 4, we proposed QueryBooster, a middleware-based service for human-centered query rewriting.

In Chapter 3, we presented the core techniques of QueryBooster. We developed a novel expressive rule language (VarSQL) for users to formulate rewriting rules easily. Furthermore, 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.

In Chapter 4, we presented the system design of QueryBooster. We used real-world applications and datasets to demonstrate the user experience of analyzing application workloads, introducing new rewriting rules to improve query performance, and sharing rewriting knowledge among users. We discussed the technical challenges and our design decisions in developing QueryBooster and showed its advantages.

5.2 Future Work

Our work has focused on middleware-based solutions to help databases to optimize SQL queries through rewriting. To decide how to rewrite an online query into a more efficient format, we have explored two directions: one is based on machine-learning models, and the other is based on human-crafted rewriting rules. We have identified several interesting research opportunities in both directions as future work.

Identifying Query Templates and Managing MDP Models Dynamically. In Chapter 2, we presented how to train an MDP model to optimize queries following a given query template. The Maliva framework assumes the developers can specify the query template for their applications. It can be even more powerful if the framework automatically identifies query templates from application workloads and starts training a new MDP model once it discovers a new query template. One approach is to adopt query clustering techniques to dynamically discover query templates by analyzing their filtering attributes and query shapes. We can maintain a set of query clusters and treat each as a query template. We can maintain an MDP model as its associated query rewriting agent for each discovered template. We can also keep the freshness of the models by adaptively adding more queries into the training set as we discover more queries clustered into the query template. In this way, the proposed Maliva framework can be more useful to practitioners.

Discovering a Rewriting Sequence from User-Given Rewriting Examples. We proposed a novel technique in Chapter 3 that can generalize a user-given rewriting pair of queries into a rewriting rule. In some cases, to achieve the user-given rewriting example, a more general solution is to apply multiple general rewriting rules sequentially instead of applying only one specific rule. In our proposed rule-suggestion framework, we assume the examples provided by the users require only one-step rewriting (i.e., only one rule is applied). If we relax this assumption, it can lower the bar for users to formulate rewriting rules through the proposed QueryBooster system. Assuming the rewriting sequence has two steps, one approach is identifying the intermediate rewritten query between the original and final rewritten queries and then generalizing two rules, respectively. One is between the original query and the intermediate rewritten query, and the other is between the intermediate and final rewritten queries. We can enumerate the number of intermediate rewritten queries for rewriting sequences with more than two steps. To identify the intermediate rewritten queries, we can expand the graph of editing the original query into the final rewritten query, where each editing operation is similar to the operations in computing the edit distance between two SQL strings. If the computational cost is high, we can explore different heuristic-based searching strategies or adopt machine-learning techniques to reduce the search space.

Using ChatGPT and Large Language Models to Improve the User Experience.

In Chapter 4, we demonstrated the user experiences that the VarSQL language and the rule-suggestion framework make the QueryBooster system very easy to use to formulate rewriting rules. As the ChatGPT service and large language models (LLM) attract significant attention from research communities, we can also consider how ChatGPT and LLMs can help improve our proposed QueryBooster system. On the one hand, ChatGPT cannot replace the online query rewriting feature of QueryBooster due to its inaccuracy in reasoning about the correctness of rewritings. Therefore, human-crafted rewriting rules are still necessary to optimize application queries. On the other hand, LLMs are proven to be good at understanding human intentions through a natural language interface. Thus, one idea is to adopt LLMs to help users formulate rewriting rules by providing rewriting examples. We can train an LLM model on data labeled by generalizing rewriting pairs of queries into rewriting rules and suggest the rewriting rules generalized by the LLM model to users.

Bibliography

[1] Scala: Quasiquotes introduction, 2023. https://docs.scala-lang.org/overviews/quasiquotes/intro.html.

[2] Andrew Kane. Dexter: The automatic indexer for postgres, 2017. https://github.com/ankane/dexter.

[3] S. C. ands Ling Xiao, J. Gerth, and P. Hanrahan. Maintaining interactivity while exploring massive time series. In *Proceedings of the IEEE Symposium on Visual Analytics Science and Technology, IEEE VAST 2008, Columbus, Ohio, USA, 19-24 October 2008*, pages 59–66. IEEE Computer Society, 2008.

[4] Apache AsterixDB, http://asterixdb.apache.org.

[5] Apache Calcite, https://calcite.apache.org/.

[6] Apache Calcite. Calcite test suite, 2021. https://github.com/georgia-tech-db/spes/blob/main/testData/calcite_tests.json.

[7] 2018. Apache Superset(incubating) - Apache Superset documentation., https://superset.incubator.apache.org/.

[8] https://www.apexsql.com/sql-tools-plan.aspx.

[9] http://asterixdb.apache.org/docs/0.9.6/sqlpp/manual.html#Query_hints.

[10] Q. Bai, S. Alsudais, C. Li, and S. Zhao. Maliva: Using machine learning to rewrite visualization queries under time constraints. In *EDBT 2023, Ioannina, Greece, March 28-31, 2023*, pages 157–170. OpenProceedings.org, 2023.

[11] https://github.com/learnedsystems/baoforpostgresql.

[12] L. Battle, R. Chang, and M. Stonebraker. Dynamic prefetching of data tiles for interactive visualization. In F. Özcan, G. Koutrika, and S. Madden, editors, *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 1363–1375. ACM, 2016.

[13] L. Battle, R. J. Crouser, A. Nakeshimana, A. Montoly, R. Chang, and M. Stonebraker. The role of latency and task complexity in predicting visual search behavior. *IEEE Trans. Vis. Comput. Graph.*, 26(1):1246–1255, 2020.

[14] F. Brauer, R. Rieger, A. Mocan, and W. M. Barczynski. Enabling information extraction by inference of regular expressions from sample entities. In C. Macdonald, I. Ounis, and I. Ruthven, editors, *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011, Glasgow, United Kingdom, October 24-28, 2011*, pages 1285–1294. ACM, 2011.

[15] M. Budiu, P. Gopalan, L. Suresh, U. Wieder, H. Kruiger, and M. K. Aguilera. Hillview: A trillion-cell spreadsheet for big data. *PVLDB*, 12(11):1442–1457, 2019.

[16] D. Cheng, P. Schretlen, N. Kronenfeld, N. Bozowsky, and W. Wright. Tile based visual analytics for twitter big data exploratory analysis. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 2–4, 2013.

[17] M. Cherniack and S. B. Zdonik. Rule languages and internal algebras for rule-based optimizers. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 401–412. ACM Press, 1996.

[18] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu. Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries. *VLDB'18*, 2018.

[19] Comby is a tool for searching and changing code structure, `https://comby.dev/`.

[20] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. Vizdom: Interactive analytics through pen and touch. *PVLDB*, 8(12):2024–2027, 2015.

[21] A. Crotty, A. Galakatos, E. Zgraggen, C. Binnig, and T. Kraska. The case for interactive data exploration accelerators (ideas). In C. Binnig, A. Fekete, and A. Nandi, editors, *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, page 11. ACM, 2016.

[22] G. Damasio, V. Corvinelli, P. Godfrey, P. Mierzejewski, A. Mihaylov, J. Szlichta, and C. Zuzarte. Guided automated learning for query workload re-optimization. *Proc. VLDB Endow.*, 12(12):2010–2021, 2019.

[23] D. Das and D. S. Batory. Praire: A rule specification framework for query optimizers. In P. S. Yu and A. L. P. Chen, editors, *Proceedings of the Eleventh International Conference on Data Engineering, March 6-10, 1995, Taipei, Taiwan*, pages 201–210. IEEE Computer Society, 1995.

[24] C. A. de Lara Pahins, S. A. Stephens, C. Scheidegger, and J. L. D. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE Trans. Vis. Comput. Graph.*, 23(1):671–680, 2017.

[25] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the*

2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016, pages 679–694, 2016.

[26] A. Eldawy, M. F. Mokbel, and C. Jonathan. Hadoopviz: A mapreduce framework for extensible visualization of big spatial data. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 601–612. IEEE Computer Society, 2016.

[27] `https://www.eversql.com/`.

[28] B. Finance and G. Gardarin. A rule-based query rewriter in an extensible DBMS. In *Proceedings of ICDE, 1991, Kobe, Japan*, pages 248–256. IEEE Computer Society, 1991.

[29] D. Fisher, I. O. Popov, S. M. Drucker, and m. c. schraefel. Trust me, i'm partially right: incremental visualization lets analysts explore large datasets faster. In *CHI Conference on Human Factors in Computing Systems, CHI '12, Austin, TX, USA - May 05 - 10, 2012*, pages 1673–1682. ACM, 2012.

[30] M. N. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: learning document type descriptors from XML document collections. *Data Min. Knowl. Discov.*, 7(1):23–56, 2003.

[31] P. Godfrey, J. Gryz, and P. Lasek. Interactive visualization of large data sets. *IEEE Trans. Knowl. Data Eng.*, 28(8):2142–2157, 2016.

[32] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Eng.*, 6(1):120–135, 1994.

[33] G. Graefe and D. J. DeWitt. The EXODUS optimizer generator. In U. Dayal and I. L. Traiger, editors, *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*, pages 160–172. ACM Press, 1987.

[34] T. Guo, K. Feng, G. Cong, and Z. Bao. Efficient selection of geospatial data on maps for interactive and visualized exploration. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 567–582. ACM, 2018.

[35] H. Hacigümüs, B. R. Iyer, C. Li, and S. Mehrotra. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the 2002 ACM SIGMOD, Madison, Wisconsin, USA, June 3-6, 2002*, pages 216–227. ACM, 2002.

[36] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Deep learning models for selectivity estimation of multi-attribute queries. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1035–1050. ACM, 2020.

[37] Y. He, X. Chu, K. Ganjam, Y. Zheng, V. R. Narasayya, and S. Chaudhuri. Transform-data-by-example (TDE): an extensible search engine for data transformations. *Proc. VLDB Endow.*, 11(10):1165–1177, 2018.

[38] https://en.wikipedia.org/wiki/Hint_(SQL).

[39] D. Hirn and T. Grust. Pgcuckoo: Laying plan eggs in postgresql's nest. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019*, pages 1929–1932. ACM, 2019.

[40] K. Z. Hu, M. A. Bakker, S. Li, T. Kraska, and C. A. Hidalgo. Vizml: A machine learning approach to visualization recommendation. In S. A. Brewster, G. Fitzpatrick, A. L. Cox, and V. Kostakos, editors, *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 128. ACM, 2019.

[41] IBM DB2 11.5: Query rewriting methods and examples, https://www.ibm.com/docs/en/db2/11.5?topic=process-query-rewriting-methods-examples.

[42] J. Im, F. G. Villegas, and M. J. McGuffin. Visreduce: Fast and responsive incremental information visualization of large datasets. In X. Hu, T. Y. Lin, V. V. Raghavan, B. W. Wah, R. A. Baeza-Yates, G. C. Fox, C. Shahabi, M. Smith, Q. Yang, R. Ghani, W. Fan, R. Lempel, and R. Nambiar, editors, *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 25–32. IEEE, 2013.

[43] Jia Yu and M. Sarwat. Accelerating spatial data visualization dashboards via a materialized sampling approach. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2020.

[44] L. Jiang, P. Rahman, and A. Nandi. Evaluating interactive data systems: Workloads, metrics, and guidelines. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1637–1644, 2018.

[45] Jinyuan Zhang and Yicun Yang. Werewriter, 2023. https://ipads.se.sjtu.edu.cn/werewriter-demo/home.

[46] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In I. F. Cruz, E. Ferrari, Y. Tao, E. Bertino, and G. Trajcevski, editors, *IEEE 30th International Conference on Data Engineering, Chicago, ICDE 2014, IL, USA, March 31 - April 4, 2014*, pages 472–483. IEEE Computer Society, 2014.

[47] Keebo: Data learning and warehouse optimization. https://keebo.ai/.

[48] K. Khurana and J. R. Haritsa. Shedding light on opaque application queries. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on*

Management of Data, Virtual Event, China, June 20-25, 2021, pages 912–924. ACM, 2021.

[49] J. Kossmann, T. Papenbrock, and F. Naumann. Data dependencies for query optimization: a survey. *VLDB J.*, 31(1):1–22, 2022.

[50] T. Kraska. Northstar: An interactive data science system. *PVLDB*, 11(12):2150–2164, 2018.

[51] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to optimize join queries with deep reinforcement learning. *CoRR*, abs/1808.03196, 2018.

[52] Kyle Lahnakoski. More sql parsing, 2023. `https://github.com/klahnakoski/mo-sql-parsing`.

[53] D. J. L. Lee and A. G. Parameswaran. The case for a visual discovery assistant: A holistic solution for accelerating visual data exploration. *IEEE Data Eng. Bull.*, 41(3):3–14, 2018.

[54] K. Li and G. Li. Approximate query processing: What is new and where to go? - A survey on approximate query processing. *Data Science and Engineering*, 3(4):379–397, 2018.

[55] L. D. Lins, J. T. Klosowski, and C. E. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Trans. Vis. Comput. Graph.*, 19(12):2456–2465, 2013.

[56] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE Trans. Vis. Comput. Graph.*, 20(12):2122–2131, 2014.

[57] Z. Liu, B. Jiang, and J. Heer. *imMens*: Real-time visual querying of big data. *Comput. Graph. Forum*, 32(3):421–430, 2013.

[58] G. Lohman. Is query optimization a “solved” problem? *ACM SIGMOD Blog.*, ACM Blog(14’), 2014.

[59] Y. Luo, C. Chai, X. Qin, N. Tang, and G. Li. Interactive cleaning for progressive visualization through composite questions. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 733–744. IEEE, 2020.

[60] Y. Luo, X. Qin, N. Tang, G. Li, and X. Wang. Deepeye: Creating good data visualizations by keyword search. In G. Das, C. M. Jermaine, and P. A. Bernstein, editors, *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1733–1736. ACM, 2018.

[61] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Learning to steer query optimizers. *CoRR*, abs/2004.03814, 2020.

[62] R. Marcus, P. Negi, H. Mao, N. Tatbul, M. Alizadeh, and T. Kraska. Bao: Making learned query optimization practical. In G. Li, Z. Li, S. Idreos, and D. Srivastava, editors, *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, pages 1275–1288. ACM, 2021.

[63] R. Marcus and O. Papaemmanouil. Deep reinforcement learning for join order enumeration. In R. Bordawekar and O. Shmueli, editors, *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 3:1–3:4. ACM, 2018.

[64] R. C. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A learned query optimizer. *Proc. VLDB Endow.*, 12(11):1705–1718, 2019.

[65] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[66] MongoDB 5.0: Query documents, `https://www.mongodb.com/docs/manual/tutorial/query-documents/`.

[67] D. Moritz, D. Fisher, B. Ding, and C. Wang. Trust, but verify: Optimistic visualizations of approximate queries for exploring big data. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems, Denver, CO, USA, May 06-11, 2017*, pages 2904–2915. ACM, 2017.

[68] D. Moritz, B. Howe, and J. Heer. Falcon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 694, 2019.

[69] B. Mozafari, J. Ramnarayan, S. Menon, Y. Mahajan, S. Chakraborty, H. Bhanawat, and K. Bachhav. Snappydata: A unified cluster for streaming, transactions and interactice analytics. In *CIDR 2017, 8th Biennial Conference on Innovative Data Systems Research, Chaminade, CA, USA, January 8-11, 2017, Online Proceedings*, 2017.

[70] MySQL 8.0: 5.6.4.2 Using the Rewriter Query Rewrite Plugin, `https://dev.mysql.com/doc/refman/8.0/en/rewriter-query-rewrite-plugin-usage.html`.

[71] `https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html`.

[72] NYC Taxi Data, `https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page`.

[73] Oracle Free Use Terms and Conditions, `https://www.oracle.com/downloads/licenses/oracle-free-license.html`.

[74] Oracle R19: 11 Basic Query Rewrite for Materialized Views, https://docs.oracle.com/en/database/oracle/oracle-database/19/dwhsg/basic-query-rewrite-materialized-views.html.

[75] https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327.

[76] Y. Park, M. J. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. In *32nd IEEE International Conference on Data Engineering, ICDE 2016, Helsinki, Finland, May 16-20, 2016*, pages 755–766. IEEE Computer Society, 2016.

[77] Y. Park, B. Mozafari, J. Sorenson, and J. Wang. Verdictdb: Universalizing approximate query processing. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1461–1476, 2018.

[78] Y. Park, S. Zhong, and B. Mozafari. Quicksel: Quick selectivity learning with mixture models. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 1017–1033. ACM, 2020.

[79] L. Parreaux, A. Voizard, A. Shaikhha, and C. E. Koch. Unifying analytic and statically-typed quasiquotes. *Proc. ACM Program. Lang.*, 2(POPL):13:1–13:33, 2018.

[80] J. Peng, D. Zhang, J. Wang, and J. Pei. AQP++: connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data, SIGMOD Conference 2018, Houston, TX, USA, June 10-15, 2018*, pages 1477–1492, 2018.

[81] H. Pirahesh, J. M. Hellerstein, and W. Hasan. Extensible/rule based query rewrite optimization in starburst. In *Proceedings of the 1992 ACM SIGMOD, San Diego, California, USA, June 2-5, 1992*, pages 39–48. ACM Press, 1992.

[82] H. Pirahesh, T. Y. C. Leung, and W. Hasan. A rule engine for query transformation in starburst and IBM DB2 C/S DBMS. In W. A. Gray and P. Larson, editors, *Proceedings of the Thirteenth International Conference on Data Engineering, April 7-11, 1997, Birmingham, UK*, pages 391–400. IEEE Computer Society, 1997.

[83] PostgreSQL 14: CREATE RULE — define a new rewrite rule, https://www.postgresql.org/docs/14/sql-createrule.html.

[84] PostgreSQL 14 Documentation: 41.2. Views and the Rule System, https://www.postgresql.org/docs/current/rules-views.html.

[85] PostgreSQL 14: Trigram index, https://www.postgresql.org/docs/current/pgtrgm.html.

[86] https://pghintplan.osdn.jp/pg_hint_plan.html.

[87] F. Psallidas and E. Wu. Provenance for interactive visualizations. In *Proceedings of the Workshop on Human-In-the-Loop Data Analytics, HILDA@SIGMOD 2018, Houston, TX, USA, June 10, 2018*, pages 9:1–9:8, 2018.

[88] QueryBooster: Improving SQL Performance Using Middleware Services for Human-Centered Query Rewriting (under review), March 2023. submitted to VLDB 2023.

[89] X. Qian, R. A. Rossi, F. Du, S. Kim, E. Koh, S. Malik, T. Y. Lee, and J. Chan. Ml-based visualization recommendation: Learning to recommend visualizations from data. *CoRR*, abs/2009.12316, 2020.

[90] Query performance insight for azure sql database. `https://docs.microsoft.com/en-us/azure/azure-sql/database/query-performance-insight-use?view=azuresql`.

[91] Query Rewrite and Optimization, `https://docs.teradata.com/r/8mHBBLGP88~HK9Auie2QvQ/4PC2qalhztpNrpq9R~zpDw`.

[92] S. Rahman, M. Aliakbarpour, H. Kong, E. Blais, K. Karahalios, A. G. Parameswaran, and R. Rubinfeld. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, 2017.

[93] V. Raman and J. M. Hellerstein. Potter's wheel: An interactive data cleaning system. In P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, editors, *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*, pages 381–390. Morgan Kaufmann, 2001.

[94] Re: How to use index in strpos function, `https://www.postgresql.org/message-id/046801c96b06%242cb14280%248613c780%24%40r%40sbcglobal.net`.

[95] regex101. regular expressions 101, 2023. `https://regex101.com/`.

[96] J. Rissanen. Modeling by shortest data description. *Autom.*, 14(5):465–471, 1978.

[97] E. A. Rundensteiner, M. O. Ward, Z. Xie, Q. Cui, C. V. Wad, D. Yang, and S. Huang. Xmdvtool^q: : quality-aware interactive data exploration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, pages 1109–1112, 2007.

[98] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach (4th Edition)*. Pearson, 2020.

[99] SAP HANA Performance Guide for Developers, `https://help.sap.com/doc/05b8cb60dfd94c82b86828ee77f7e0d9/2.0.04/en-US/SAP_HANA_Performance_Developer_Guide_en.pdf`.

[100] E. Sciore and J. S. Jr. A modular query optimizer generator. In *Proceedings of the Sixth International Conference on Data Engineering, February 5-9, 1990, Los Angeles, California, USA*, pages 146–153. IEEE Computer Society, 1990.

[101] S. Sikdar and C. Jermaine. MONSOON: multi-step optimization and execution of queries with partially obscured predicates. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 225–240. ACM, 2020.

[102] Snowflake documentation, `https://docs.snowflake.com/en/index.html`.

[103] Software is fragile - Missing source code, on a massive scale, `https://www.softwareheritage.org/mission/software-is-fragile/`.

[104] SQL Server 2019: SQL Server technical documentation, `https://docs.microsoft.com/en-us/sql/sql-server/?view=sql-server-ver15`.

[105] `https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql-query?view=sql-server-ver15`.

[106] V. Sreekanti, C. Wu, X. C. Lin, J. Schleier-Smith, J. Gonzalez, J. M. Hellerstein, and A. Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[107] J. Sun and G. Li. An end-to-end learning-based cost estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[108] R. S. Sutton and A. G. Barto. *Reinforcement learning - an introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[109] Tableau, https://www.tableau.com/.

[110] W. Tao, X. Liu, Y. Wang, L. Battle, Ç. Demiralp, R. Chang, and M. Stonebraker. Kyrix: Interactive pan/zoom visualizations at scale. *Comput. Graph. Forum*, 38(3):529–540, 2019.

[111] Toad: Develop, analyze, and administer databases with toad. `https://www.toadworld.com/products`.

[112] M. Tokic. Adaptive epsilon-greedy exploration in reinforcement learning based on value difference. In R. Dillmann, J. Beyerer, U. D. Hanebeck, and T. Schultz, editors, *KI 2010: Advances in Artificial Intelligence, 33rd Annual German Conference on AI, Karlsruhe, Germany, September 21-24, 2010. Proceedings*, volume 6359 of *Lecture Notes in Computer Science*, pages 203–210. Springer, 2010.

[113] TPC-H Website, `http://www.tpc.org/tpch/`.

[114] TPC-H Website, `http://www.tpc.org/tpch/`.

[115] I. Trummer, J. Wang, D. Maram, S. Moseley, S. Jo, and J. Antonakakis. Skinnerdb: Regret-bounded query evaluation via reinforcement learning. In P. A. Boncz, S. Manegold, A. Ailamaki, A. Deshpande, and T. Kraska, editors, *Proceedings of the 2019*

International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019, pages 1153–1170. ACM, 2019.

[116] A. Vogelsgesang, M. Haubenschild, and et al. Get real: How benchmarks fail to represent the real world. In *Proceedings of DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 1:1–1:6. ACM, 2018.

[117] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. *PVLDB*, 9(3):84–95, 2015.

[118] Q. Wang, Z. Chen, Y. Wang, and H. Qu. Applying machine learning advances to data visualization: A survey on ML4VIS. *CoRR*, abs/2012.00467, 2020.

[119] Y. Wang, K. Feng, X. Chu, J. Zhang, C. Fu, M. Sedlmair, X. Yu, and B. Chen. A perception-driven approach to supervised dimensionality reduction for visualization. *IEEE Trans. Vis. Comput. Graph.*, 24(5):1828–1840, 2018.

[120] Z. Wang, N. Ferreira, Y. Wei, A. S. Bhaskar, and C. Scheidegger. Gaussian cubes: Real-time modeling for visual exploration of large multidimensional datasets. *IEEE Trans. Vis. Comput. Graph.*, 23(1):681–690, 2017.

[121] Z. Wang, Z. Zhou, and et al. Wetune: Automatic discovery and verification of query rewrite rules. In *SIGMOD '22, Philadelphia, PA, USA, June 12 - 17, 2022*, pages 94–107. ACM, 2022.

[122] C. J. C. H. Watkins and P. Dayan. Technical note q-learning. *Mach. Learn.*, 8:279–292, 1992.

[123] Wikipedia contributors. Regular expression, 2022. `https://en.wikipedia.org/wiki/Regular_expression`.

[124] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In C. S. Jensen, C. M. Jermaine, and X. Zhou, editors, *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, pages 1081–1092. IEEE Computer Society, 2013.

[125] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty Aware Query Execution Time Prediction. *PVLDB*, 7(14):1857–1868, 2014.

[126] J. Yu, R. Moraffah, and M. Sarwat. Hippo in action: Scalable indexing of a billion new york city taxi trips and beyond. In *33rd IEEE International Conference on Data Engineering, ICDE 2017, San Diego, CA, USA, April 19-22, 2017*, pages 1413–1414. IEEE Computer Society, 2017.

[127] J. Yu, Z. Zhang, and M. Sarwat. Geosparkviz: a scalable geospatial data visualization framework in the apache spark ecosystem. In D. Sacharidis, J. Gamper, and M. H. Böhlen, editors, *Proceedings of the 30th International Conference on Scientific and Statistical Database Management, SSDBM 2018, Bozen-Bolzano, Italy, July 09-11, 2018*, pages 15:1–15:12. ACM, 2018.

[128] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20-24, 2020*, pages 1297–1308. IEEE, 2020.

[129] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 913–918, 2015.

[130] X. Zhang, J. Wang, J. Yin, and S. Ji. Sapprox: Enabling efficient and accurate approximations on sub-datasets with distribution-aware online sampling. *PVLDB*, 10(3):109–120, 2016.

[131] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *Proc. VLDB Endow.*, 15(1):46–58, 2021.