UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

MACHINE LEARNING ENABLED QUERY RE-OPTIMIZATION ALGORITHMS FOR CLOUD DATABASE SYSTEMS

A DISSERTATION

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

DOCTOR OF PHILOSOPHY

By

CHENXIAO WANG
Norman, Oklahoma
2021

MACHINE LEARNING ENABLED QUERY RE-OPTIMIZATION ALGORITHMS
FOR CLOUD DATABASE SYSTEMS

A DISSERTATION APPROVED FOR THE
SCHOOL OF COMPUTER SCIENCE

BY THE COMMITTEE CONSISTING OF

Dr. Le Gruenwald, Chair

Dr. Mohammed Atiquzzaman

Dr. Qi Cheng

Dr. Sudarshan Dhall

Dr. Xiangming Xiao

© Copyright by CHENXIAO WANG 2021
All Rights Reserved.

ACKNOWLEDGEMENTS

This work would have been impossible to be accomplished all by my own. Many people have provided their support, encouragement to me during these years. First of all, I would like to express my sincere gratitude to my dissertation committee chair, Dr. Le Gruenwald, for her expert advice, patience, and supervision throughout my Ph.D. studies. Also, I would like to extend my thanks to my dissertation committee members, Dr. Mohammed Atiquzzaman, Dr. Qi Cheng, Dr. Sudarshan Dhall, and Dr. Xiangming Xiao, for their invaluable feedback and guidance. Also, I would like to thank Dr. Laurent d'Orazio for his collaboration on this research and the members of The University of Oklahoma's Database Group (OUDB) for their suggestions and inspiration.

Moreover, I would like to express my special thanks to the National Science Foundation and the University College at The University of Oklahoma for their financial support for my studies. Also, Assistant Dean, Dr. Johnnie-Margaret McConnell, and Assistant to the Director, Ms. Ingrid Ter Steege have provided me with a lot of support while I was working for them in the past years.

Finally, I am deeply indebted to my family and my friends. Although I cannot stay with them, they are always standing behind me.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS .... **iv**
LIST OF FIGURES .... **viii**
LIST OF TABLES .... **x**
ABSTRACT .... **xi**
CHAPTER I INTRODUCTION **1**
1.1 The Problem of Query Re-Optimization in Cloud DBMS 1
1.2 Background 2
1.2.1 Query Optimization and Re-Optimization in Traditional DBMS 2
1.2.2 Cloud DBMS and Query Re-Optimization 6
1.3 Objective 9
1.4 Contribution 10
1.5 Organization 12
CHAPTER II LITERATURE REVIEW 13
2.1 Query Re-Optimization Algorithms for Cloud Database Systems 13
2.1.1 Rule-based Re-Optimization 14
2.1.2 Stage-based Re-Optimization 15
2.1.3 Sample-based Re-Optimization 17
2.1.4 Resource Provisioning-based Query Re-Optimization 19
2.2 Query Re-Optimization Algorithms for Cloud Database Systems Using Machine Learning Techniques 20
2.2.1 Re-Optimization Using a Reinforcement Learning Model 20
2.3 Summary 26
CHAPTER III A PROPOSED QUERY RE-OPTIMIZATION ALGORITHM FOR CLOUD DATABASE SYSTEMS (ReOpt) **28**
3.1 Motivation of ReOpt 28
3.2 Overview of ReOpt 31
3.3 Details of ReOpt 31

3.4 Summary 38
CHAPTER IV A PROPOSED MACHINE LEARNING BASED QUERY RE-OPTIMIZATION ALGORITHM (ReOptML)... 40
4.1 Motivation of ReOptML 41
4.1.1 Supervised Learning-based Algorithms for Query Re-Optimization 44
4.1.2 Unsupervised Learning-based Algorithms for Query Re-Optimization 44
4.2 Overview of ReOptML 45
4.3 Details of ReOptML 47
4.3.1 Training Data Collection and Feature Selection 51
4.3.2 Machine Learning Model Selection 53
4.3.3 Applying Supervised Learning Model to Query Re-Optimization 54
4.4 Summary 60
CHAPTER V PROPOSED REINFORCEMENT LEARNING BASED QUERY RE-OPTIMIZATION ALGORITHMS FOR CLOUD DATABASE SYSTEMS (ReOptRL and SLAReOptRL) 62
5.1 Reinforcement Learning-Based Algorithms for Query Re-Optimization 63
5.2 Motivation of ReOptRL 63
5.3 Overview of ReOptRL 66
5.4 Details of ReOptRL 67
5.5 Reward Function 72
5.6 SLA-Aware Reinforcement Learning-based Algorithms for Query Re-Optimization (SLAReOptRL) 74
5.6.1 SLA Definition 74
5.6.2 Extending ReOptRL to Consider SLA Violation 75
5.7 Summary 77
CHAPTER VI PERFORMANCE ANALYSIS 78
6.1 Theoretical Analysis 78
6.1.1 Proof of Correctness of ReOpt, ReOptML and ReOptRL 78
6.1.2 Time Complexity Analysis of ReOpt, ReOptML and ReOptRL 90
6.2 Experimental Results 105

6.2.1 Experimental Hardware and Software Configurations and Benchmark Dataset 105
6.2.2 Competitive Algorithms 106
6.2.3 Performance Metrics 109
6.2.4 Experimental Results for ReOpt 111
6.2.5 Experimental Results of ReOptML 116
6.2.6 Experimental Results of ReOptRL and SLAReOptRL 125
6.3 Summary 136
CHAPTER VII CONCLUSIONS AND FUTURE WORK 138
7.1 Summaries of Performance Evaluation Results 140
7.1.1 Summary of Performance Results of ReOpt 140
7.1.2 Summary of Performance Results of ReOptML 141
7.1.3 Summary of Performance Results of ReOptRL and SLAReOptRL 142
7.2 Future Research 142
REFERENCES 144

LIST OF FIGURES

Figure 1. Steps of query processing 4
Figure 2. Query processing using ReOpt 33
Figure 3. Dispatch function....34
Figure 4. Optimization function....35
Figure 5. Query processing using ReOptML....48
Figure 6. The procedure for collecting training data 51
Figure 7. Sample query 51
Figure 8. QEP is divided into different stages after being compiled from the query 52
Figure 9. Example of MergeTable 55
Figure 10. Query processing algorithm with machine learning-based re-optimization....59
Figure 11. Merge and GenerateQEP function....60
Figure 12. General procedure of reinforcement learning....63
Figure 13. Procedures of Q-Learning (the top figure) and DQN (the bottom figure) 67
Figure 14. Procedure of ReOptRL 70
Figure 15. Query processing using reinforcement learning-based re-optimization....72
Figure 16. QEP P_1 generated by the query optimizer before re-optimization 82
Figure 17. QEP P_2 after 1st re-optimization 84
Figure 18. QEP P_2 after operator O_1 is merged 84
Figure 19. QEP P_2 after operators O_1 and O_2 are merged....85
Figure 20. QEP P_2 after 2nd re-optimization 86
Figure 21. QEP P_2 after operators O_1 and O_3 are merged....87
Figure 22. Impact of data size on query response time of Query 1 112
Figure 23. Impact of data size on query monetary cost of Query 1 113
Figure 24. Impacts of data size on time for executing query....116
Figure 25. Impacts of data size on monetary cost for executing query 116
Figure 26. Model accuracy of three different machine learning algorithms that learn from queries executed on (a) uniform data and (b) skewed data 118
Figure 27. (a) and (b) Average response time and average monetary cost of executing queries using three different machine learning models for query re-optimization....121
Figure 28. (a)-(d) Average query response time and monetary cost of executing one query from different query types on skew data (a-b) and on uniform data (c-d) 125
Figure 29. Time performance for executing queries using different algorithms 129
Figure 30. Monetary cost performance for executing queries using different algorithms 129

Figure 31. Average SLA violation rate when executing queries using different algorithms 130
Figure 32. The impacts of RatioJOIN on the query execution time improvement when queries are executed using different re-optimation algorithms compared with NoReOpt..132
Figure 33. (a) and (b) Impacts of the weight of time on the performance improvement of the re-optimization algorithms over the baseline algorithm "NoReOpt"...134
Figure 34. (a) and (b) Impacts of the weight of monetary cost on the performance improvement of the re-optimization algorithms over the baseline algorithm "NoReOpt" 135

LIST OF TABLES

Table 1. Feature comparison of the query re-optimization techniques for cloud database systems 27
Table 2. List of selected features 53
Table 3. Sequence of searching matching operators in MergeTable 57
Table 4. Line by line time cost of ReOpt 92
Table 5. Line by line time cost of ReOptML 95
Table 6. Line by line time cost of ReOptRL 103
Table 7. The comparison of the peak number of containers used in execution of Query 1 114
Table 8. Average and cumulative query response time and monetary cost using three different machine learning models 120
Table 9. Performance results (Average Values ± Standard Deviations) of different algorithms. The number (x) after each reported value indicates the ranking of the algorithm with rank (1) being the best 137

ABSTRACT

In cloud database systems, hardware configurations, data usage, and workload allocations are continuously changing. These changes make it difficult for the query optimizer to obtain an optimal query execution plan (QEP) for a query based on the data statistics collected before the query execution. In order to optimize a query with a more accurate cost estimation to achieve such a QEP, performing query re-optimizations during the query execution has been proposed in the literature. However, some of the re-optimizations may not provide any gain in terms of query response time or monetary cost and may also have negative impacts on the query performance due to their overheads. This raises the question of how to determine when a re-optimization is beneficial. In addition, a Service Level Agreement (SLA) is signed between users and the cloud. Thus, query re-optimization is multi-objective optimization that minimizes not only query execution time and monetary cost but also SLA violation. However, none of the existing query re-optimization algorithms considers all these three objectives together and none of them can predict when a re-optimization is beneficial.

To fill the gap, in this dissertation, four novel query re-optimization algorithms, ReOpt, ReOptML, ReOptRL and SLAReOptRL are proposed. Extensive theoretical and experimental evaluations performed on our proposed techniques showed that each of them

has better performance in terms of time, monetary cost, and SLA violation rate than state-of-the-art techniques when applied to the TPC-H database benchmark.

CHAPTER I
INTRODUCTION

1.1 The Problem of Query Re-Optimization in Cloud DBMS

In optimizing a query for fast execution, a traditional database management system (DBMS) through its query optimizer is expected to produce an optimal query execution plan (QEP) to execute the query. A popular way for the DBMS to derive this QEP is to estimate the query cost using the available statistics of the database. However, as the database changes over time, the statistics available at the time when the QEP is derived may not reflect the actual database statistics during the query execution, and thus the QEP may not be optimal. To solve this problem, query re-optimization conducted during query execution has been proposed for traditional DBMS.

Query re-optimization is more challenging in cloud DBMS due to the dynamic nature of cloud environments, the monetary costs that users have to pay to cloud service providers, and the service level agreements (SLAs) between tenants and cloud providers, which if violated, cloud providers have to pay penalties. While query re-optimization in traditional DBMS only needs to deal with query response time, query re-optimization in cloud DBMS needs to deal with all three performance objectives: query response time, monetary costs, and SLA violation. However, none of the existing query re-optimization techniques for cloud DBMS address all these objectives together. By using the existing techniques that do not consider monetary costs, such as [1, 2, 3, 4], users may be charged with a large amount of money for executing queries, or by using the existing techniques that do not

consider SLA violation, such as [5], cloud providers may be heavily penalized. In addition, none of the existing techniques can predict when a query re-optimization is beneficial to conduct so that the overheads incurred by unnecessary query re-optimizations can be reduced. Thus, the existing techniques may not provide any gain in terms of query response time or monetary cost and may also have negative impacts on the query performance due to their overheads [1, 2, 3]. It is therefore important to develop a query re-optimization algorithm for cloud DBMS that can address all the above issues.

1.2 Background

In this section, we provide some background concepts that are necessary for the reader to follow the ideas introduced later in this dissertation. Section 1.2.1 gives a brief introduction to query optimization and re-optimization in traditional database systems. Section 1.2.2 introduces query re-optimization in cloud database systems.

1.2.1 Query Optimization and Re-Optimization in Traditional DBMS

In this section, we present the background of traditional DBMS and its query optimization and re-optimization processes.

1.2.1.1 Database System and Query

According to the definition given in [8], "A database is an organized collection of structured information, or data, typically stored electronically in a computer system. A database is usually controlled by a database management system (DBMS)." The data here

refer to anything that is digitalized, such as a text file, an image, or a clip of video. The database is where those data are placed. The software that manages these data is called a DBMS. Not like a file management system, DBMS organizes stored data in a specific data structure and provides users with an interface to access and modify data more efficiently. Besides that, a modern DBMS also has more functionalities in addition to storing data. Data restoration, replication, protection, etc. are also important features of today's DBMS.

Query in general means a request to retrieve data from a database system. To let DBMS communicate with users easily, a structured query language (SQL) is invented in the late 1970s. SQL is a standard programming language for using relational database management systems. SQL or SQL-like programming language is still widely popular and accepted by most of the DBMS on the market today. Thus, a query usually refers to a query written in SQL or a SQL-like programming language unless stated otherwise.

1.2.1.2 Query Optimization in Traditional DBMS

The first DBMS was invented in the 1960s by IBM [9] and has been evolved for many decades. Many concepts and notable products are developed from that time till today, such as System R (the late 1970s), Oracle (1980s), MySQL (1990s), and NoSQL (2000s). There are different types of DBMS products. In this dissertation, to distinguish other database systems from a cloud database system, we call the products that do not use any cloud techniques as traditional DBMS or traditional database systems in the following chapters.

In Section 1.2.1.1, we mentioned that the purpose of a query is for the user to communicate with the DBMS. To let the DBMS understand the query, the high-level query language queries have to be translated into low-level expressions. Those expressions are then translated into machine-readable codes and the codes are executed in the end. This process is called query processing. Query processing is one of the most key processes happened in a DBMS. Figure 1 shows the major steps of query processing in a relational DBMS.

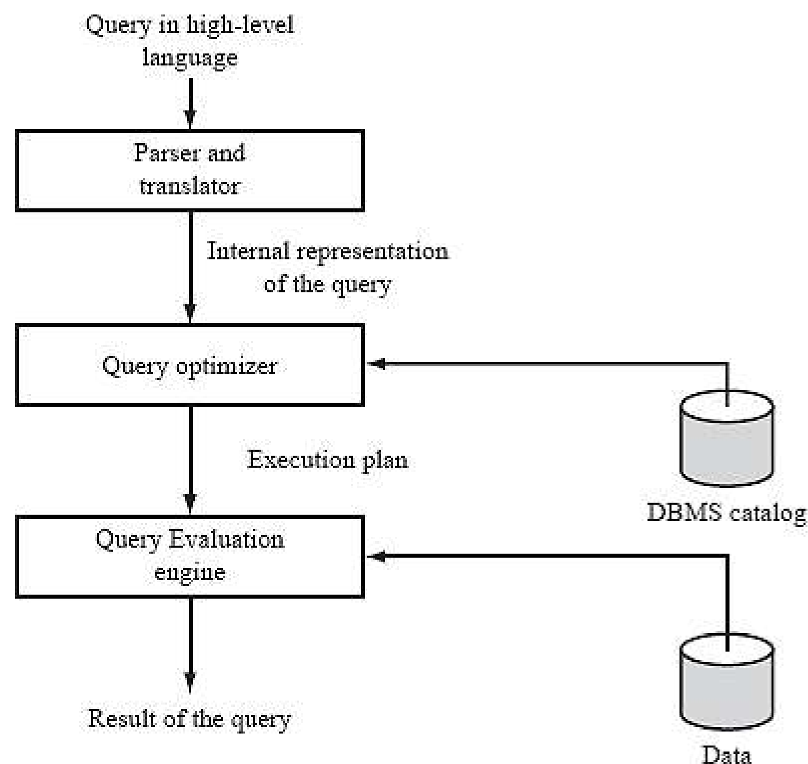


Figure 1. Steps of query processing [9]

After a query is received by the DBMS, it is checked for syntax and compiled to a relational algebra representation by the parser and translator. The sequences of performing the operators in the relation algebra form logical trees. Then, the query optimizer converts this representation to physical query execution plans. Each physical query execution plan is

evaluated to find its estimated cost for execution. The query optimizer uses its cost model to do this evaluation and this evaluation usually works with the meta-data of the attributes and tables stored in the DBMS, such as the selectivity of an attribute, average row size of a tuple, etc. Those meta-data are referred to as data statistics. After all the query execution plans are evaluated, the query execution plan with the best cost is selected. Notice that different query optimizers have their definition of the "best query execution plan". Some optimizer considers the query execution plan that has the fastest response time as the optimal plan while the others may select the query execution plan that uses the least hardware resources. After a query execution plan is chosen, it is converted to low-level machine code and executed on the stored data. Finally, the results are given to the user after the execution.

1.2.1.3 Query Re-Optimization in Traditional DBMS

Query re-optimization means executing a portion of an optimized query execution plan, measuring the data statistics, and optimizing the plan again before continuing execution [10]. The runtime data statistics are collected after a portion of an optimized query execution plan has been executed for updating the estimated data statistics. Those updated data statistics are used by the query optimizer to re-optimize the remainder of the partially executed query execution plan. As a result, in the new query execution plan, the optimizer may choose different join orders, join algorithms, and/or execution order of query operators based on the new data statistics. Such query re-optimization usually happens multiple times during the entire query execution for the best performance.

1.2.2 Cloud DBMS and Query Re-Optimization

In this section, we introduce cloud DBMS and its query re-optimization.

1.2.2.1 What Is a Cloud DBMS?

A cloud database system is a database system built and deployed on a cloud platform, such as Amazon AWS [11] and Microsoft Azure [12]. A cloud database system is usually provided as a service called Database as a Service (DaaS). Users access such a database system via the interface provided by the service providers. A cloud database system serves many of the same major functions as a traditional database system. It provides persistent storage and enables users to add, update, modify and delete data through provided APIs. In addition, it adds cloud computing features, such as high availability and scalability [13, 14]. Also, it is a fee-based subscription service in which the database runs on the service provider's infrastructure. There is a minimum requirement needed for the user to maintain and manage the system.

Based on those advantages and the requirements of today's applications, many IT companies, and academic institutions focus on researching and developing cloud database system products. Popular products, such as Amazon RDS, Oracle Cloud Database, and Microsoft Azure Database are widely used in many applications.

1.2.2.2 Why Using a Cloud DBMS?

From the user's perspective, the benefits of using a cloud database system include the following:

(a) Freedom from administration and configuration

All the major products serve as a black box and the users can access it using the provided GUI, command-line interface, or APIs. There is no need for the users to hire domain experts to install the software, tune the system parameters, or monitor the status of the running system. It is always "ready to use" whenever the users have access to the internet.

(b) Freedom from physical hardware

The users do not need to consider how much system resource they need to purchase for running the system. Making a plan of how many machines and what kind of machines is not an easy task. Purchasing not enough machines may result in a system that does not meet requirements. While over-purchasing wastes a lot of money. On a cloud database system, there is no need to worry about these issues. The cloud provider owns all the infrastructure. The users can access to that infrastructure whenever they need it via the internet.

(c) Easy to scale the database system

With the usage changes of the users' applications, it is very common for the users to add in or remove some resources. On a cloud database system, this can be done easily by typing several lines of commands via the interface.

(d) Monetary cost savings

Users are provided with a “pay-as-you-go” style of charges. They only need to pay for the amount of resource they used. Usually, the price is charged whenever applications are running. Monetary costs can be saved while applications are idle. In addition, due to the advantages mentioned in (a), no need to hire experts also saves monetary costs.

(e) High availability for the database system

The database system runs on a highly reliable platform. When a user provisions a database instance, the database system synchronously replicates the data to a standby database instance which is generally in a different availability zone or data center. The database system also performs backups, snapshots, and host replacement automatically. All these tasks make the database highly available and durable.

1.2.2.3 Query Re-Optimization in Cloud DBMS

In a cloud database system, query re-optimization is still an important feature. Query re-optimization in cloud DBMS shares the same mechanism as in traditional DBMS. At the beginning, many efforts [15, 16, 17, 18, 2] were made to improve query re-optimization to reduce query response time only. This is known as single-objective query re-optimization. Those techniques adapt query re-optimization algorithms in traditional DBMS [19, 20] to the cloud environment. In traditional DBMS, query re-optimization only has one objective, which is query response time. Later, cloud service providers have implemented a pay-as-you-go price model for charging their services. In this price model, users only pay for the

services as long as they use the services without requiring long-term contracts [11]. Moreover, cloud service providers sign an agreement with their users before the users purchase their services. In the agreement, the cloud providers commit to providing the quality of their services and claim penalties if they fail to fulfill it. This agreement is also known as a service level agreement (SLA). Under this scenario, some recent query re-optimization techniques in cloud DBMS [7, 6] consider multi-objectives for optimization. However, besides query response time, they consider only either monetary cost [6] or SLA violation [7]. None of the existing query re-optimization algorithms consider all these three objectives together. In addition, the query execution performances of these techniques suffer from the overheads caused by conducting unnecessary query re-optimizations as they cannot predict whether a re-optimization is beneficial before conducting it.

1.3 Objective

The objective of this research is to develop a novel query re-optimization technique for cloud database systems that has the following abilities:

- Ability to re-optimize a query execution plan taking query response time, monetary cost, and SLA violation into consideration simultaneously.
- Ability to predict whether a query re-optimization is beneficial to be conducted after a query operator or a stage of query operators is executed in order to avoid unnecessary query re-optimizations.
- Ability to re-optimize a query execution plan without depending on the accuracy of the data statistics collected by the DBMS.

1.4 Contribution

To fill the gaps stated in Section 1.1, in this dissertation, we propose four different query re-optimization algorithms for a cloud DBMS. These four algorithms are stage-based query re-optimization (ReOpt), query re-optimization using machine learning (ReOptML), query re-optimization using reinforcement learning (ReOptRL), and SLA-aware query re-optimization using reinforcement learning (SLAReOptRL).

The first algorithm introduced in this work is ReOpt [21]. It is a query processing algorithm in a cloud database system that does multi-objective query re-optimization. In this algorithm, the query execution plan is optimized not only to reduce the query response time but also to reduce the monetary cost needed to execute the query.

The second algorithm introduced is ReOptML [22]. The goal of designing the second algorithm is to address one major issue found in the ReOpt. The issue is that a lot of the query re-optimizations conducted are not necessary. The reason is that the re-optimization does not always happen at the best timing during the query execution. Doing unnecessary query re-optimizations adds extra overheads. To reduce the number of unnecessary query re-optimizations, in ReOptML, we train a supervised learning model to predict whether the re-optimization is useful or not. Only useful re-optimizations are allowed to be conducted afterward.

The third algorithm introduced in this work is ReOptRL [23]. The purpose of designing this algorithm is to address two issues found in ReOpt and ReOptML. The first issue is that they both require updated data statistics whenever a re-optimization happens. Updating data statistics does help the query optimizer generate a better query execution plan. However, this operation itself is very expensive. The second issue is that with ReOptML, the training data with the labels indicating which historical cases needed re-optimizations and which historical cases did not need re-optimizations must be available to train the supervised learning model. To avoid depending on the updated data statistics in query re-optimization and on the availability of the training data, the reinforcement learning technique is used in ReOptRL. In this technique, the query re-optimization process does not require any data statistics and training data, and the learning model alone decides how to optimize the query execution plan. Moreover, nowadays, since SLA is a very important feature specifically for multi-tenant cloud platforms, ReOptRL is further extended to SLAReOptRL, a technique that aims to reduce SLA violations in addition to query response time and monetary cost in re-optimizing queries.

To the best of our knowledge, there does not exist a query re-optimization technique for a cloud DBMS that considers query response time, monetary cost, and SLA violation at the same time; predicts whether a re-optimization is beneficial; and does not depend on data statistics. Our proposed techniques fill the gap.

For performance studies, we provide both comprehensive theoretical and experimental analyses of the proposed algorithms. In theoretical analysis, we present the worst-case time complexity and correctness proofs of proposed algorithms. In experimental analysis, we present the studies of the proposed algorithms in comparison with the state-of-the-art techniques. The results show that in most cases, our proposed techniques outperform existing techniques.

1.5 Organization

The rest of the dissertation is organized as follows. Chapter II reviews the existing work related to query processing for a cloud DBMS. Chapter III describes ReOpt, our proposed technique for cloud database query re-optimization. Chapter IV describes ReOptML, our proposed technique for supervising learning-based query re-optimization on cloud databases. Chapter V describes ReOptRL and SLAReOptRL, our proposed techniques for reinforcement learning-based query re-optimization on cloud databases. Chapter VI presents the analytical results as well as the experimental results studying the performance of our proposed techniques. Finally, Chapter VII provides conclusions and future research directions.

CHAPTER II
LITERATURE REVIEW

The problem of query re-optimization has been studied in the literature. In the early days, heuristics were used to decide when to re-optimize a query or how to do the re-optimization. Usually, these heuristics were based on cost estimations which were not accurate at the time when query re-optimization takes place. Besides that, sometimes, a human-in-the-loop was needed in order to analyze and adjust these heuristics [24, 15]. These add additional overheads caused by query re-optimization to the overall performance of queries. Unfortunately, these heuristic solutions can often miss good query execution plans. More importantly, traditional query optimizers rely on static strategies, and hence do not learn from previous experience. Traditional systems plan a query, execute the query execution plan, and forget they ever optimized this query. Because of the lack of feedback, a query optimizer may select the same bad plan repeatedly, never learning from its previous bad or good choices.

2.1 Query Re-Optimization Algorithms for Cloud Database Systems

In this section, we present a brief survey of some of the query processing techniques on cloud database systems that use re-optimization. These techniques are aiming to address the problems of query processing on cloud database systems raised in Chapter I.

2.1.1 Rule-based Re-Optimization

Rule-based re-optimization techniques re-optimize the query execution plan based on one or several human-determined rules. Whenever the monitored status of the query execution meets the rule(s), the re-optimization is triggered.

In the early stage, progressing query optimization (POP) [18, 17, 15] is used. It detects cardinality errors in the middle of execution. The actual runtime cardinality is compared to the estimated cardinality. If there is a large difference between them, then the re-optimization is triggered. This technique is originally designed for a centralized database system. Later, Stillger *et al.* [2] have integrated this idea in the "LEO-DB2", which is a similar technique that re-optimizes queries on a cloud database system.

In these techniques, the inputs are a query q and a threshold t; they proceed as follows:

1. Query q is compiled and converted into a query execution plan P by an existing query optimizer.
2. Plan P is sent to the execution engine and paused at some check points set by the human. Usually, those check points are placed after certain types of query operator.
3. Data statistics are updated. The actual cardinality of each attribute in the participated tables is checked. If the following holds,

$$\frac{|cumulative\ estimated\ cardinaltiy - cumulative\ actual\ cardinality|}{cumulative\ estimated\ cardinaltiy} > t$$

then the re-optimization is triggered, and the rest of the query execution plan is re-optimized again by the same query optimizer.

Advantages:

- The actual data statistics are used to re-optimize the unfinished part of the query execution plan. Also, cardinality change is an important indicator to tell whether the rest of the query execution plan has the potential to be improved by re-optimization.
- The re-optimization decision can be efficiently made as the decision-making only depends on one rule.

Disadvantages:

- One main disadvantage about this technique is that the location of the check points is still decided by a human. These check points might not align with the best timing to do the re-optimization.
- Another disadvantage is that the threshold is fixed. Using a fixed value is very hard to adapt this technique to a different application built on a dynamic environment.

2.1.2 Stage-based Re-Optimization

More recently, Bruno *et al.* [1] have proposed a query optimization method during the query execution. The query execution is monitored and paused multiple times at the point of one stage of the operators' finished execution. Those stages are determined by the query

optimizer. Similar to the previous algorithm, at each of the points, a new estimation of executing the rest part of the query is made with statistics collected from the finished query, and the rest of the query that has not been executed is adjusted with the new estimations. The adjusted query applies more accurate estimations so that the query performance is improved.

This technique accepts a query *q* as its input and proceeds as follows:

1. Query *q* is compiled and converted into a query execution plan *P* by an existing query optimizer. Also, plan *P* is divided into different stages.
2. The first stage of operators is sent for execution. The actual data statistics are collected and used to update the current data statistics.
3. The rest of the query execution plan is re-optimized using the updated data statistics.
4. The current query execution plan is merged with the original query execution plan.
5. The next stage of the operators is continuously executed until all the operators are executed.

Advantages:

- The re-optimization is triggered whenever one stage of operators finished execution. By doing this, more re-optimization is triggered compared to the previous algorithm.

- The query execution plan after re-optimization is combined with the original query execution plan so that there are minimal changes in the original plan.

Disadvantages:

- The main disadvantage of this technique is that it heavily depends on the stage divided by the query optimizer. If the optimizer fails to divide the stage well, the actual timing of triggering the re-optimization still cannot be aligned with the best timing to do the re-optimization

2.1.3 Sample-based Re-Optimization

Nikolay *et al.* [25] have proposed EARL, a query re-optimization technique specially designed for the Hadoop system. It evaluates a sample size of stored data and adjusts the cost estimation by the results of the sample evaluation. To process a sample-based re-optimization, this technique takes query q, sample size w as input, and proceeds as follows:

1. Query q is compiled and converted into a query execution plan P by an existing query optimizer.
2. The execution of plan P is converted into Jobs and the Jobs are assigned to Workers. This process is determined by the Hadoop system.
3. After one Worker finishes execution, a selected data set sample of size w is compared to the same data set before execution. A covariance function is used to calculate the difference between distribution of the sample data set and the entire data.

4. This difference is used to adjust the cost estimation and adjust the rest of the plan *P* using the updated cost estimation.

Advantages:

- This technique uses the evaluation by one sample dataset to update the cost estimation. It saves a large amount of time as getting the actual data statistics from the entire data set is an expensive operation.

Disadvantages:

- Using a sample dataset saves time overhead, but it is not as accurate as getting the actual data statistics from the entire data.
- The timing of re-optimization is fixed which only happens after one Worker finishes execution. Again, by doing this, the timing of re-optimization may not be aligned with the best timing of re-optimization.

Wu *et al.* [3] have proposed Sample, another query re-optimization algorithm that updates data statistics estimated from a sample

of tuples collected during the runtime. This algorithm takes Query *q* and sample size *w* as input and proceeds as follows:

1. Query *q* is compiled and converted into a query execution plan *P* by an existing query optimizer.
2. The first available operator in the query execution plan *P* is executed.

3. After a sample size *w* of tuples are processed, the cardinalities of the columns in the table are updated by the cardinality of sample.
4. The rest of query execution plan *P* is re-optimized using the updated cardinality.
5. Steps 2 to 4 are repeated until all the operators finish.

Advantages:

- The time of updating data statistics is short because the query optimizer only needs to update the data statistics collected from a sample dataset.

Disadvantages:

- In this algorithm, the query re-optimization only utilizes the updated cardinalities. Other data statistics, such as average tuple size, histogram, and the number of tuples, are not updated. Using partial data statistics in query re-optimization may still produce a sub-optimal query execution plan.

2.1.4 Resource Provisioning-based Query Re-Optimization

Costa *et al.* [7] have proposed a query re-optimization algorithm that focuses on resource provisioning. There are different types of nodes in the system. The algorithm sends parts of the query to a node of one type and measures the tuple read rate, which estimates time in seconds for the node to process a quantity of tuples. If the tuple read rate exceeds its estimated value, the next part of the plan is sent to a node of another type.

Advantages:

- This algorithm also considers SLA violation when selecting the type of node to execute a query.
- This algorithm can be built on an existing query optimizer without modification.

Disadvantages:

- This algorithm only re-optimizes the hardware resource allocation; it does not adjust the query execution plan.

2.2 Query Re-Optimization Algorithms for Cloud Database Systems Using Machine Learning Techniques

In Section 2.1, query re-optimization algorithms without machine learning techniques are surveyed. Although the query execution plans are improved after re-optimization using the surveyed techniques, they still suffer from different problems. One major problem is that they rely on human-tuned heuristics for different purposes. To help improve the accuracy of re-optimization and reduce the overhead of doing re-optimization, machine learning techniques are adapted to query re-optimization. In the following sections, several query re-optimization algorithms using machine learning techniques are surveyed.

2.2.1 Re-Optimization Using a Reinforcement Learning Model

In this section, several query re-optimization algorithms using reinforcement learning algorithms are surveyed. Reinforcement learning is an online model and does not require

any training dataset. It can learn to improve query processing by running more queries through trials and errors.

2.2.1.1 SkinnerDB

Trummer *et al.* [5] have proposed an algorithm using the regrets-bounded model to adjust the join order of the query execution plan. This algorithm also re-optimizes the query execution plan in the middle of query execution. After a batch of tuples is executed for a join operator, the table that the next batch of tuples joins is adjusted based on the decision made by the model. The choice is evaluated by a reward and this reward is used to adjust the model. By doing this, this model becomes more accurate with more execution of the join operator. This algorithm receives a query *q* and tuple batch size *b* as input. Notice that, in this technique, the query *q* must be an SJP query. This algorithm then proceeds as follows:

1. Query *q* is compiled and converted into a query execution plan. This plan is executed till the Join operator.
2. Using the Upper Confidence Bounds for Trees (UCT) to pick the first two tables A and B that participate in the Join. UCT is an algorithm that applies bandit ideas to guide Monte-Carlo planning [26].
3. The Join operator executes the first batch size *b* of tuples from table A and pauses.
4. The Reward (Regrets) is computed for this execution.
5. Using the UCT algorithm again to select the next table to participate in the Join for the next batch size of tuples.

6. Steps 1 to 5 are repeated until all the tuples are joined.

Advantages:

- This technique can adjust the Join order at a fine granulated level. The Join order is adjusted after a batch size of tuples is executed. This means more tuples benefit from the optimal Join order.
- Using the UCT algorithm guarantees that there is both exploitation and exploration when searching for the next table to be joined.

Disadvantages:

- This technique can only optimize SPJ queries and assumes all the joins are left-deep joins. It cannot deal with more complex query types and bushy joins.
- Only the Join order is optimized. The other part of the query execution plan and the resource provision is not included in this algorithm.

2.2.1.2 ReJoin

Marcus *et al.* [6] have proposed a technique that uses deep reinforcement learning to re-optimize queries. In this technique, a query is encoded with a one-hot vector and is then sent to a deep neural network (DNN). The output of the DNN produces a probability distribution over potential action. Those actions are choosing which tables to participate in the join. A reward is also calculated after the selected action is performed. This reward is

sent back to improve the DNN. Given a query *q* and available relations $\{r_1, r_2, r_3 \ldots\ldots r_n\}$ as input, this technique proceeds as follows:

1. Query *q* is compiled and converted into a query execution plan *P* by an existing query optimizer.
2. The Join operator in the query execution plan *P* is then converted into a one-hot vector form which is called the state matrix *m*. This state matrix m uses a special format to present the attributes from different relations that participate in a Join operation.
3. The state matrix *m* is sent to the reinforcement learning model to decide the next action that the query optimizer should take. In this technique, the model is a deep neural network (DNN), and the actions are the potential join orders that the query execution plan would take.
4. Each action is evaluated and an argmax function is used to select the best action.
5. The execution engine performs the selected action and gives feedback through the reward. In this technique, the reward for every non-terminal state (a partial ordering) is zero, and the reward for an action arriving at a terminal state S_f (a complete ordering) is the reciprocal of the cost of the join.
6. This reward is also used by the DNN to adjust the weights.

Advantages:

- The Join order optimization is independent of the query optimizer. The reinforcement learning model decides the join order. The join order is better than the one optimized by the original query optimizer if the model is well trained.

- The traditional query optimizer does not learn from past queries because it lacks feedback. Thus, a bad query execution plan might be chosen repeatedly. This technique addresses this issue.

Disadvantages:

- Still, this technique only focuses on join order enumeration. The other types of query operators and resource provision optimization are not investigated.

2.2.1.3 CuttleFish

Kaftan *et. al* [4] have proposed a technique that uses a reinforcement learning model to tune the join operator. In this technique, the multi-armed bandit (MAB) model is used to decide the best physical operator to implement the join. This technique takes query q as input and proceeds as follows:

1. Query q is compiled and converted into a query execution plan P by an existing query optimizer.
2. The query execution plan P is executed and is paused if a join operator is encountered.

3. The Multi-armed bandit (MAB) model is used to decide the best physical operator of this join. Inside MAB, instead of using ε-greedy, Thompson sampling randomly chooses arms according to the likelihood that they have the highest expected reward.
4. This join is then executed using the selected physical operator.
5. The execution of query execution plan P continues until the next join operator is encountered.
6. Steps 2 to 5 are repeated until the execution of query execution plan P finishes.

Advantages:

- This algorithm does not require updated data statistics for choosing the best physical join operator.
- Adapting Thompson sampling to this algorithm guarantees that the action with a high reward is selected.

Disadvantages:

- This algorithm only focuses on selecting the best physical operator of join. Other types of query operators and resource provision optimization are not investigated.

Table 1 presents a feature comparison of the query re-optimization techniques for cloud database systems reviewed in Chapter II. A cell containing the word “Yes” means that the technique referred to in that row addresses the issue listed as the header of that

column, and a cell containing “No” means the technique does not address the corresponding issue.

2.3 Summary

In this chapter, we surveyed existing query re-optimization techniques in cloud DBMS. As shown in Table 1, none of the surveyed techniques addresses all the issues. In particular, none of the surveyed techniques has considered query response time, monetary cost, and SLA violation simultaneously, and none of them can predict whether a re-optimization is beneficial before conducting it. Additionally, only ReJoin [6] and SkinnerDB [5] do not require updated data statistics and ReJoin [6] re-optimizes queries in offline mode. Moreover, ReJoin [6], Cuttlefish [4], and SkinnerDB [5] do not target re-optimizing the whole query execution plan and the technique proposed by Stillger *et al.* [2] still needs human interference. To fill the gaps in the literature, we introduce our four algorithms in the next four chapters. First, in Chapter III, we introduce ReOpt, the first query re-optimization algorithm in cloud DBMS that considers both query response time and monetary costs. Then, we introduce ReOptML to address the issue of deciding whether re-optimization is beneficial in Chapter IV. Finally, we introduce ReOptML to address the issue of depending on updated data statistics in query re-optimization and SLAReOptRL to consider SLA violation in Chapter V.

Table 1. Feature comparison of the query re-optimization techniques for cloud database systems

	Multi-objectives			**Requiring Updated Data Statistics**	**Deciding Whether Re-Optimization is Beneficial**	**Processing Mode**	**End-to-End Query Optimization**	**Without Human Interference**
	Time	Money	SLA					
Stillger (2003) [2]	Yes	No	No	Yes	No	Online	Yes	No
Bruno (2013) [1]	Yes	No	No	Yes	No	Online	Yes	Yes
EARL (2012) [25] &Sample (2016) [3]	Yes	No	No	Yes	No	Online	Yes	Yes
Costa (2016) [7]	Yes	No	Yes	Yes	No	Online	Yes	Yes
CuttleFish (2018) [4]	Yes	No	No	Yes	No	Online	No	Yes
SkinnerDB (2018) [5]	Yes	No	No	No	No	Online	No	Yes
ReJoin (2018) [6]	Yes	Yes	No	No	No	Offline	No	Yes
ReOpt (2018) [21]	Yes	Yes	No	Yes	No	Online	Yes	Yes
ReOptML (2020) [22]	Yes	Yes	No	Yes	Yes	Offline	Yes	Yes
ReOptRL (2021) [23]	Yes	Yes	No	No	Yes	Online	Yes	Yes
SLAReOptRL (2021) [23]	Yes	Yes	Yes	No	Yes	Online	Yes	Yes

CHAPTER III
A PROPOSED QUERY RE-OPTIMIZATION ALGORITHM FOR CLOUD DATABASE SYSTEMS (ReOpt)

Most of the existing database query optimization techniques are designed to target traditional database systems with only one optimization objective. These optimization algorithms are not suitable for cloud database systems. Users will take both query response time and monetary cost paid to the cloud service providers into consideration for selecting a database system product. Thus, query optimization for cloud database systems needs to target reducing monetary cost in addition to query response time. This means that query optimization is more challenging than one objective found in traditional algorithms.

We present a novel stage-based query re-optimization algorithm for cloud database systems (ReOpt) in the following sections of this chapter. In Section 3.1, the motivation of ReOptL is introduced. In Section 3.2, the overview of ReOpt is given. In Section 3.3, we present the details of ReOpt.

3.1 Motivation of ReOpt

Query optimization on a cloud database differs from optimization on a traditional distributed database for several reasons. First, a cloud database is provided to the user via a leasing service with several options of payment. The user would need to take the monetary cost paid to the cloud service provider for query processing into consideration on top of the query response time. While in traditional database query optimization, the monetary cost is usually negligible because the infrastructure configuration is fixed, and

the monetary cost is paid up-front. Thus, in the usage of a cloud database system, the user can provide both the query response time limit and monetary budget of a query, which are defined as User Constraints. Query optimization becomes multi-objectives to satisfy multiple user constraints. Secondly, a cloud database is elastic. Cloud service providers provide a finite pool of virtualized on-demand resources. Similarly, users can decide the number and types of containers on which they would like to run their queries, and they can change the combination of container types over time. If users select more containers or more powerful containers, the time cost of the query execution may decrease, but the monetary cost may increase. That is, the time cost often contradicts the monetary cost. Query optimization on cloud databases should balance both time and monetary cost so that the users can obtain the result of the query with all the user constraints being satisfied. So, cloud database systems are responsible for providing the users with a feasible query optimization solution to deliver the query results that satisfy the user constraints as well as minimize the multiple costs of query execution. Besides that, the time and monetary costs needed to execute a query are estimated based on the data statistics that the query optimizer has available when the query optimization is performed. These statistics are often not accurate, which may result in inaccurate estimates for the time and monetary costs needed to execute the query. Thus, the query execution plan (QEP) generated before the query is executed may not be the best one. Adaptively optimizing the QEP during the query execution to employ more accurate statistics will yield better QEP selection, and thus will improve query performance. There are some existing techniques that address part of these issues, which is that the selected QEP is suboptimal. However, they do not optimize queries

based on both time and monetary costs and do not take adaptive optimization into consideration [27].

Optimizing a query in a cloud database environment requires an important consideration. Since the query will be executed on multiple nodes, one must consider how to allocate computational resources optimally as there is an infinite number of workload/node combinations. However, not all allocation solutions are feasible. Users will have query constraints, and resource allocation will influence performance. Thus, optimal resource allocation becomes a problem, known simply as the scheduling problem. A scheduling algorithm is also applied for resource allocation on cloud systems. Besides that, good data statistics are critical to deriving a good schedule. They will affect the overall performance of query execution as a sub-optimal query execution schedule will be produced by the optimizer if data statistics are erroneous. An effective schedule is based on accurate cost calculations of the tasks to be scheduled. It would be beneficial if we could use the actual runtime query statistics instead of their estimates in the query optimization process. This is because estimates may not be as accurate as the actual running statistics. However, existing techniques [28, 29] either focus on optimizing queries based on only time, which is not sufficient for cloud database environments or do not consider query re-optimization for more accurate statistics.

3.2 Overview of ReOpt

In this technique, a regular query optimizer first generates an initial QEP. Then this QEP will be divided into stages and executed by the execution engine stage by stage. After finishing each stage, the data statistics will be updated. These statistics include the cardinality, selectivity, and max and min values for each attribute in each database table. By updating these statistics, the estimation of the resulting data size used in the next stages will be updated accordingly. The rest of the stages in the QEP are also sent to the query optimizer for re-optimization using the updated statistics. Three things are required to be submitted to the system by the user: a query, a time constraint, and a monetary cost constraint. Our adaptive optimization algorithm (ReOpt) presented in Figure 2 is the main framework that gives an overview of how the query is processed. Algorithm 1 in Figure 2 will call the algorithms in Figures 3 and 4. Algorithm 2 in Figure 3 describes how the containers are assigned to execute the QEP and Algorithm 3 describes how each schedule is optimized individually.

3.3 Details of ReOpt

As we can see in Figure 2, the user submits a query and the time and monetary cost constraints for finishing the query. In Line 1, the query is compiled into a query optimizer tree. This tree contains all the physical operators needed to process the query. Line 2 shows that these operators are grouped into different stages. The operators that do not require the results from their previous operators can be grouped. In Line 3, the Optimizer_Tree is processed one stage at a time. In Line 4, a stage is mapped to a DAG according to their

dependencies in the Optimizer_Tree. In Line 5, Algorithm 2 is called to generate an initial schedule which is optimized by Algorithm 3 in Line 6. The result is then obtained by executing all the operators in the current stage according to the Optimized_Schedule. The finished operators in the current stage are then eliminated from the Optimizer_Tree. The process from Line 3 to Line 9 is repeated for each stage until all the stages are finished and the result is returned to the user. The following paragraph explains how to find an optimized schedule and illustrates the cost-reducing re-optimization process of this schedule. To better illustrate the idea, we provide a running example.

```
SELECT Department, count(Name)
FROM STUDENT
GROUP BY Department
WHERE Grade <='C';
```

Suppose we execute the above query, and the user constraints are as follows: the query response time must be less than 2 minutes, and the monetary cost must be less than \$30. Assume that each container costs \$0.1 per second. The database table STUDENT is stored in 3 separate locations. Each table has three columns, Name, Department and Grade, and each of the three tables contains 65,000 rows of data. The first step is converting the query to the optimized operator tree like in traditional database systems. The query optimizer groups these operators including TableScan, Filter, Sort, Aggregation, Merge, and Partition into different stages. After the stages are formed, the first operator TableScan will be executed on 3 data partitions in parallel on 3 different containers and the allocation of containers is decided by the Algorithm 2. In Algorithm 2, the execution time of each operator executed on each container is first estimated. Then, from Line 6 to Line 9, a set of

candidate containers are found for the next operator that has no dependencies. These candidate containers are the ones that the operation execution time estimate satisfies the user time constraint. From Line 13 to 17, this operator will be assigned to the container which has the shortest estimation time. This assignment is then added to the schedule with its current timestamp as the starting time. The current timestamp plus the estimating execution time is added as the ending time.

Algorithm 1: ADAPTIVE OPTIMIZATION (ReOpt)

INPUT:
Sql: query
CONS: two-dimensional variable containing time and money constraints.
C: a set of containers each of which has the percentage of the current CPU usage and the network bandwidth.
P: unit price of leasing one container.
Min_value: a loop control parameter.
Iter_limit: a pre-defined variable.
OUTPUT:
Result: the result of the query.

1. Ops ← compile query Sql to get its set of compiler-generated operators
2. Optimizer-Tree ← generate a multi-staged optimizer tree from the set of operators Ops
3. **for each** stage in the multi-staged Optimizer-Tree
4. G ← map the stage in Optimizer-Tree to form a dataflow graph
5. Initial-Schedule ← call function DISPATCH (G, C, CONS) to assign operators to containers to form the initial schedule
6. Optimized-Schedule ← call function OPTIMIZE (Initial-schedule, CONS, Min_value, Iter_limit, P) to find the optimized schedule for the initial schedule
7. Result ← execute the current stage of Optimized-Schedule
8. Optimizer-Tree ← Eliminate the finished operators from the Optimizer-Tree
9. Update constraints and data statistics
10. **end for**
11. **return** Result

Figure 2. Query processing using ReOpt

Algorithm 2: DISPATCH

INPUT:
G: the dataflow graph.
C: a set of containers.
CONS: two-dimensional variable containing time and money constraints.
OUTPUT:
SG: Schedule with assignment of operators to container.

1. SG. assigns ← Ø
2. ready ← {operators in G have no dependencies}
3. **for** all operators in dataflow graph G
4. estimation_duration ← {estimate execution time of each operator}
5. **end for**
6. **while** ready! = Ø do
7. n ← {Next operator to assign}
8. candidates ← {containers that assignment of n satisfy CONS}
9. **if** candidates = Ø then
10. **return** ERROR
11. **else**
12. C ← {the container which has minimum time cost if this operator run on this container}
13. Assign (n, C)
14. ready ← ready - {n}
15. ready ← ready + {operator that have no dependencies}
16. start_time ← {current timestamp}
17. SG.assigns ← SG.assigns + {assign (n, c, start_time, start_time+estimation_duration)}
18. **end while**
19. **return** SG

Figure 3. Dispatch function

Algorithm 3: OPTIMIZE (OPT)

INPUT:
Initial_schedule: a schedule to be optimized.
Cons: a two-dimensional variable containing time and money constraints.
Minimum value: a loop control parameter.
Iteration_limit: a pre-defined variable.
P: unit price of leasing one container.
OUTPUT:
SG: an optimized schedule with estimated time and money costs that satisfies the constraints

```
1.  old_schedule ← Initial_schedule
2.  old_cost ←  GET_COST (old_schedule, P)
3.  while T is greater than Minimum value
4.     while i is less than Iteration_limit
5.        new_schedule ← {find a neighbor schedule of old_schedule}
6.        new_cost ← GET_COST (new_schedule, P)
7.     if new_cost dominates old_cost a new_cost satisfies Cons
8.        add the new_schedule to the schedule space
9.        old_schedule ← new_schedule
       else
10.       ap ← {calculate the acceptance probability with old_cost, new_cost and T}
11.         if ap is greater than a multi-dimension value in every dimension
12.            old_schedule ← new_schedule
13.         end if
14.    end if
15.     i++
16.   end while
17. reduce the value of T
18. end while
19. return SG ← {select a schedule from the schedule space}
```

FUNCTION: GET_COST (Schedule, P)

INPUT:
Schedule: a schedule needs to be evaluated for the cost.
P: unit price of leasing one container.
OUTPUT:
Cost: a two-dimensional variable contains time and monetary costs of the input schedule.

```
1.  Cost.time ← Ø
2.  Cost.money ← Ø
3.  for each assignment A in Schedule
4.    if A.t_end is the largest timestamp
5.        Cost.time ← A.t_end
6.    end if
7.  Cost.money ← Cost.money + (A.t_end - A.t_start) * P
8.  end for
9.  return Cost
```

Figure 4. Optimization function

This process keeps repeating until all the operators in the DAG have been assigned. Since this schedule is not optimized yet, it is called the initial schedule. One initial schedule looks like the following:

initial_schedule =
{
Assign(SOR_1^1,c1,12.3,75)
Assign(TS_1^1,c1,0,12.3)
Assign(TS_2^1,c2,0,65)
Assign(SOR_2^1,c3,12.3,65)
Assign(FIL_1^1,c1,75,75.05)
Assign(FIL_1^1,c2,75,75.05)
}

where Assign(SOR_1^1,c1,0,75) means the sort operator $SOR_1^1$is assigned to be executed on container 1, the estimated starting time is 0 and the estimated ending time is 75. This initial schedule may not meet the constraints, so it will then be optimized by the simulation annealing algorithm [30] presented in Algorithm 3. This creates an optimized schedule that satisfies user constraints. The following is an example of an optimized schedule. We can see that the assignment of TS_1^1 is changed from c1 to c3

optimized_schedule = {
Assign(SOR_1^1,c1,12.3,75)
Assign(TS_1^1,c3,0,12.3)
Assign(TS_2^1,c2,0,65)
Assign(SOR_2^1,c3,12.3,65)
Assign(FIL_1^1,c1,75,75.05)
Assign(FIL_1^1,c2,75,75.05)
}

From the optimized schedule above, we obtain the estimated total time for executing the query as 75.05 as the last operator finished at 75.05 seconds and the monetary cost is calculated by each container cost $0.1 per second which is

$$\left(\frac{\$0.1}{container * s}\right)(75.05\ s)(3\ containers) = \$22.515$$

After the completion of a stage, the statistics are updated with the new statistics collected from the finished stage. The user's constraints will be adjusted to reflect the remaining constraints for the unfinished stages. Each new constraint is computed as follows:

$$\text{New Constraint} = \text{Old Constraint} - (\text{Elapsed Cost} + \text{Overhead})$$

where the Elapsed Cost is the accumulated actual time and monetary cost of all the previously executed stages and the Overhead is the overhead of collecting the new statistics and updating the estimations. For example, after the execution of Stage 1, we update the constraints first as follows:

$$New\ Time\ Constraint = 120\ s\ - (75.05\ s\ + 0.01\ s) = 44.94\ s$$

$$New\ Money\ Constraint = \$30 - (\$22.515 + \$0.003) = \$7.482$$

Then we update the operators in the unfinished stages with the new statistics gathered from the completed stages. For example, the actual data size after executing the FIL operators in Stage 1 is lower than the estimated data size before the query is executed. Then, the number of containers needed to execute the AGG operator in Stage 2 is reduced; accordingly, otherwise, the number of containers used in Stage 2 is not updated and there will be some wasted containers. In our example, the number of containers needed to execute Stage 2 is

reduced from 2 to 1. Thus, the total monetary cost is reduced. Using the same procedure, this AGG operator will still be sent to the scheduler to be optimized and executed.

An optimized schedule of Stage 2 is:

optimized_schedule=
{
Assign(AGG_1^2,c1,75.05,75.10)
}

Suppose the time cost of finishing the AGG operator is 0.05 sec. In the original schedule, the monetary cost of finishing Stage 2 is

$$(2\ containers)\left(\frac{\$0.1}{container * s}\right)(0.05\ s) = \$0.01$$

and after the query re-optimization, even the time remains unchanged, but the monetary cost becomes

$$(1\ container)\left(\frac{\$0.1}{container * s}\right)(0.05\ s) = \$0.005$$

For this partial query, the monetary cost is halved. This will benefit the total time cost as well as the monetary cost of the whole query execution plan. Such savings are substantial considering the high number of queries issued in many real-world applications.

3.4 Summary

In this chapter, we presented the query re-optimization algorithm, ReOpt. In this algorithm,

after one query operator or a stage of query operators has been executed, the updated data statistics are used by the query optimizer to re-optimize the remainder of the query execution plan, considering both query response time and monetary cost. In the next chapter, to reduce unnecessary query re-optimizations, we propose the second algorithm, ReOptML.

CHAPTER IV
A PROPOSED MACHINE LEARNING BASED QUERY RE-OPTIMIZATION ALGORITHM (ReOptML)

In cloud environments, hardware configurations, data usage, and workload allocations are continuously changing. These changes make it difficult for the query optimizer to select an optimal query execution plan. To optimize a query with more accurate cost estimation, performing query re-optimizations during the query execution has been proposed in the literature [18]. However, some of the re-optimizations may not provide any gain in terms of query response time or monetary costs, which are the two optimization objectives for cloud databases, and may also have negative impacts on the performance due to their overheads. This raises the question of how to determine when a re-optimization is beneficial. In this chapter, we present a technique that uses machine learning-based re-optimization that executes a query in stages, predicts whether a query re-optimization is beneficial after a stage is executed, and invokes the query optimizer to perform such re-optimization automatically.

We present ReOptML in the following sections of this chapter. In Section 4.1, the motivation of ReOptML is introduced. In Section 4.2 and Section 4.3, the overview and details of ReOptML are given.

4.1 Motivation of ReOptML

One key difference between query optimization in cloud databases and in conventional databases is that query optimization in cloud databases seeks to reduce the monetary cost paid to cloud service providers in addition to the query response time. The time and monetary costs needed to execute a query are estimated based on the data statistics available to the query optimizer at the moment when the query optimization is performed. These statistics are often approximate, which may result in inaccurate estimates for the time and monetary costs needed to execute the query [14]. Thus, query execution plans generated before query execution may not be the best.

One approach that can be applied to address the previously mentioned issue is adaptive query processing [15]. This strategy consists of not executing queries as a whole at one time, but instead dividing the execution of each query into multiple stages and then re-running the query optimizer after each stage is executed. By doing this, the query optimizer can collect more accurate statistics in-between stage executions, which may allow for changing the QEP at runtime, thus possibly improving query performance [18]. Operators that do not rely on the completion of others are grouped and such groups are called "Stages". For example, if a QEP has a join operator, its left and right sides are each executed in a separate stage. After the completion of each stage of the QEP, the data statistics are updated, so that the query optimizer can make use of the latest statistics to generate improved (i.e.re-optimized) QEPs for those stages that remain to be executed. As a result of query re-optimization, the QEPs of stages that have not yet been executed may

change because the operators in these QEPs might be replaced by others, or because any stage might be re-scheduled to run on a different machine. Such changes in QEPs might produce different query response times and different monetary costs. However, calling the query optimizer multiple times during query execution has an associated time overhead, which in turn produces additional monetary costs. For this reason, it is desirable to re-optimize a query only if the cost improvements of the re-optimized QEP over the original QEP can offset the cost incurred in calling the optimizer multiple times.

At any given stage of the execution of a query, deciding if a re-optimization will likely bring performance improvements is not an easy task. In early work [19], such a decision is made by a rule-based heuristic. Several check points are placed manually between a certain type of operator. The difference between the estimated cost and the actual cost of executing the query after a check point is reviewed. If such difference exceeds a pre-defined threshold, then re-optimization takes place. The problem with this technique is that the rule of placing check points and the threshold is fixed. Due to the dynamic of the cloud environment, the timing of re-optimization decided by this technique is not accurate enough to reduce the query execution time. The work in [1] presents a query processing algorithm that performs query re-optimization after the completion of each stage. However, that work shows that many of these re-optimization calls produced no change in the underlying QEP, which means that the query re-optimization was performed unnecessarily. This was because the stages were not aligned with the best timing to apply the re-optimization. For example, after running the example Query 1 given in Section 6.2.4.1, we

observed that out of the 10 times that the optimizer was called for re-optimization during the execution of this query, only 2 out of these calls changed the QEP for the remaining stages; Therefore, the majority of the re-optimization calls produced no improvement on either the time or the monetary cost.

Naturally, calling the re-optimization routine unnecessarily increases both the query response time and monetary cost. The problem, therefore, lies in determining the most appropriate time when to call for re-optimization, and in determining those occasions where re-optimization can negatively impact query performance. To address this problem, this chapter presents a new machine learning-based algorithm for query re-optimization in the cloud. The key idea behind this algorithm consists in using past query executions to learn to predict the effectiveness of query re-optimizations, and this is done to help the query optimizer avoid unnecessary query re-optimizations for future queries. While machine learning has been used to improve query processing in recent work, such as [31, 32], they have not been used to avoid unnecessary query re-optimization calls in adaptive query processing.

Among the issues that need to be addressed when using machine learning for this purpose are the following. The first one consists of the many features that influence query cost estimations, such as selectivity, cardinality, min and max values of a column, the most frequent value of a column, histogram, etc. The difficulty here lies in selecting the most

appropriate subset out of all these features. The second issue consists of the large space of possible machine learning models.

4.1.1 Supervised Learning-based Algorithms for Query Re-Optimization

Supervised learning algorithms like Random Forest and Support Vector Machine [33] are suitable but need to be used correctly. The common issue of using a machine learning model is about the collection of the historical data on the selected subset of features that are needed to train the prediction model constructed using the selected machine learning algorithm. Specifically, for supervised learning models, the training data also need to be labeled. Labeling this data requires a lot of efforts and sometimes this task is not doable when the size of the dataset is large.

4.1.2 Unsupervised Learning-based Algorithms for Query Re-Optimization

To avoid the effort that needs to be put into data labeling, unsupervised learning-based models are used, such as clustering and neural network. In those models, they allow the model to work on its own to discover information that was previously undetected, and they deal with the unlabeled data. However, the problem of applying an unsupervised learning model is that the user still needs to define the classes after the data is sorted into some pattern and also the accuracy of using an unsupervised learning model is lower than using a supervised learning model [33].

Thus, measuring the effectiveness of the learning algorithm becomes a research problem. Some works such as [27] show the learning algorithm is effective for their own purposes, such as improving the cost estimation, but not all of them are effective in actual query execution performance. The selection of the machine learning model in this algorithm is discussed in Section 4.3.

4.2 Overview of ReOptML

To provide more details to support our motivation for the work proposed in this chapter, in this section we report the findings we obtained when performing query re-optimization without using machine learning. We discovered that query re-optimization can enable the optimizer to select better physical operators to execute the QEP and select better hardware configurations to execute the QEP (such as the number of containers and the type of containers). Also, in our system, multiple machines with different hardware configurations are used in parallel to execute query operators. Our best QEP considers not only the query response time but also the monetary cost. In order to take both of them into consideration, we use the Normalized Weighted Sum Model [34] to select the best plan. The idea is that every possible QEP alternative is rated by a score that combines both the objectives, time, and monetary costs, with the weights defined by the user and the environment for each objective, and the user-defined acceptable maximum value for each objective. The following function is used to compute the score of a QEP:

$$A_i^{WSM-score} = \sum_{j=1}^{n} w_j \frac{a_{ij}}{m_j} \quad (1)$$

a_{ij} is the value of alternative i (QEP_i) for objective j, m_j the user-defined acceptable maximum value for objective j, and w_j the normalized composite weight of user and environment for objective j is defined as follows:

$$w_j = \frac{uw_j * ew_j}{\sum(uw * ew)} \quad (2)$$

where uw_j and ew_j describe the weight of the user and the environmental weight for objective j, respectively. The user weight is from the user's input. Since the different objectives are representative of different costs, the algorithm chooses the alternative with the lowest score to minimize costs.

These optimizations are beneficial for improving either the overall query execution time or the monetary cost or both. In our experiment query, which is Query 1 shown in Section 6.2.4.1, there is a join of two subqueries. The data size of each subquery is unknown. We want to see how the physical operator of this join will change depending on the data size of the subquery. So, we purposely make the data size of the right side of the join operator small enough to fit in the cache. As a consequence, the Shuffle Join operator is changed to the Broadcast Join operator only after the re-optimization. Broadcast Join is executed around 40% faster than Shuffle Join in our experiments. The results show that using re-optimization

has approximately 20% improvement on average in terms of the overall time cost over using no re-optimization, while the monetary costs of the two approaches are close, with only a 4% difference. This increase in monetary cost is because the more powerful containers that are selected to run the query are the containers that charge more hourly.

If the query is re-optimized only when such changes can be guaranteed, there will not be any unnecessary re-optimization. To detect such changes, in the next section, we present a new machine learning-based technique to predict if a QEP will change after a re-optimization based on the historical query execution data is performed.

In the next section, the four parts of this algorithm are presented: feature selection, training data collection, machine learning model selection, and the query processing algorithm that integrates with the machine learning-based re-optimization to optimize query response time and monetary cost.

4.3 Details of ReOptML

Figure 5 shows the major steps of our proposed ReOptML algorithm. First, the optimizer receives a query and records the current data statistics. Then the query is compiled into a QEP with the stage information. The first stage in the QEP is executed and removed from the QEP. During execution, the data statistics are monitored and updated. After the execution of the first stage, these updated data statistics are compared with the current data statistics that were recorded before the stage was executed. The supervised learning model is used here to make the difference between the current data statistics and the new data

statistics as input and record the re-optimization decision ("YES" or "NO") as output. The query is re-optimized if the decision is "YES" and the current first stage in the new QEP after the re-optimization is executed; otherwise, if the decision is "NO", the QEP remains the same and its next stage is executed. This procedure continues until there is no stage left.

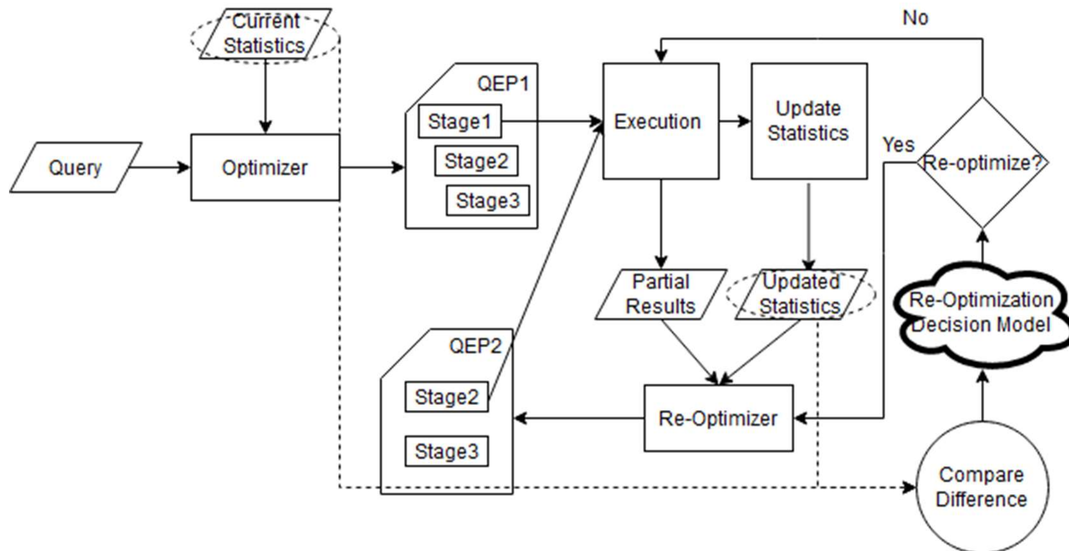


Figure 5. Query processing using ReOptML

The changes in a QEP after a re-optimization implies such re-optimization is beneficial. We define such changes occurred on a QEP if at least one of the following occurs: 1) changes in the physical operator types, 2) changes in the number of containers, or 3) changes in the types of containers. This means that if any of these three changes occurs, then re-optimization should take place.

1) A change in the physical operator types means that if there exists any physical operator in the current QEP that is different from the physical operators in the

previous QEP, then the QEP has changed. For example, in our previous experiments, the change in the physical operator from Shuffle Join to Broadcast Join is defined as a change in the physical operator types. This change highly influences query execution time. Thus, by detecting such changes in the QEP after a re-optimization, this re-optimization will probably be beneficial, and thus the re-optimization will be applied if a similar situation is encountered.

2) A change in the number or types of containers means that the total number of containers used to execute the current QEP is different from that of the previous QEP. Such changes are also called changes in the degree of parallelism. For example, the TableScan operator is assigned to four containers before the re-optimization and uses only three containers after the re-optimization. This change highly influences the monetary cost of query execution. Thus, such re-optimization becomes useful if such changes are detected.

3) Similarly, a change in the types of containers means that after the re-optimization, the operators are assigned to different types of containers than the ones that the operators were assigned to before the re-optimization. These new containers may be more or less powerful than the old ones. Detecting such changes may influence the monetary cost as well.

The above three changes occur whenever the estimated data size has also changed. This is because the query optimizer uses these estimations to decide how to execute the query and

how many containers should be used. Thus, in order to tell whether the re-optimization will be beneficial, we use the data features that are relevant to the changes in data size estimation.

Assume that in the current DBMS, there exist the C_1, C_2, …, C_n columns in all the tables. The differences in the selectivity (DIFF_SELECTIVITY), in the number of distinct values (DIFF_NDV), and in the histograms (DIFF_HISTOGRAM) of each column before and after a stage is executed are used as the data features in the training data used for prediction. The binary value YES/NO is used as the predicted class in the training data, where YES means that the re-optimization is predicted to be useful and NO otherwise. Many works show that the selectivity, the number of distinct values, and the histogram influence the data size estimation [13, 35]. Thus, the differences in these three features before and after a stage is executed result in changes in the data size estimation of the intermediate results. Hence, they become relevant in deciding the effectiveness of re-optimization. This model is applicable to the database system which has these features available.

4.3.1 Training Data Collection and Feature Selection

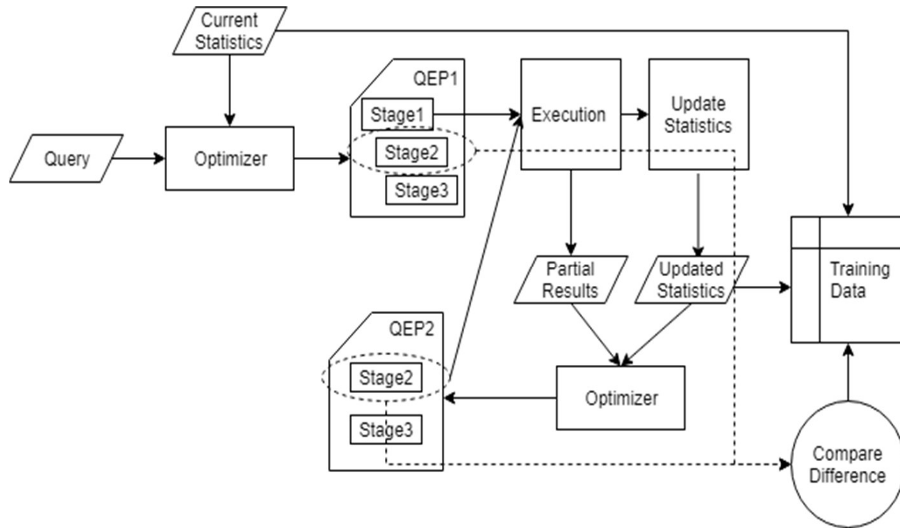


Figure 6. The procedure for collecting training data

First, we collect the training data by running random queries generated from all 22 types of queries in the TPC-H benchmark [36] on our system and recording the data statistics, which are the values of the features we have selected in Section 4.3.1. This way the prediction model can be applied to all queries. If re-optimization is only for the costliest/most representative queries, then in this first step, the training data should be collected from running only the random but most costly/representative queries.

```
SELECT Department, COUNT(Name)
FROM STUDENT
GROUP BY Department
WHERE Grade <= 'C'
```

Figure 7. Sample query

Figure 6 shows the procedure of the training data collection. In order to better explain in detail how the training data is collected, we demonstrate an example of executing the

following sample query shown in Figure 7. After the query is submitted, we record the current data statistics gathered from the system logs. These current statistics are called $Stat_{curr}$. Then, the query is sent to the optimizer to generate a QEP. This QEP includes the stage information and the nodes on which these stages will be executed. Figure 8 shows the QEP generated by the query optimizer for the sample query. In Figure 8, each node stands for a query operator. The arrows indicate the data flow between the operators. The QEP is divided into stages, each of which is denoted by a rectangular. TS, SOR, FIL, and AGG stand for TableScan, Sort, Filter, and Aggregate operators, respectively. In a cloud database system, as data are distributed among different containers, the subscripts distinguish the same operators that are executed in parallel on different data on different containers.

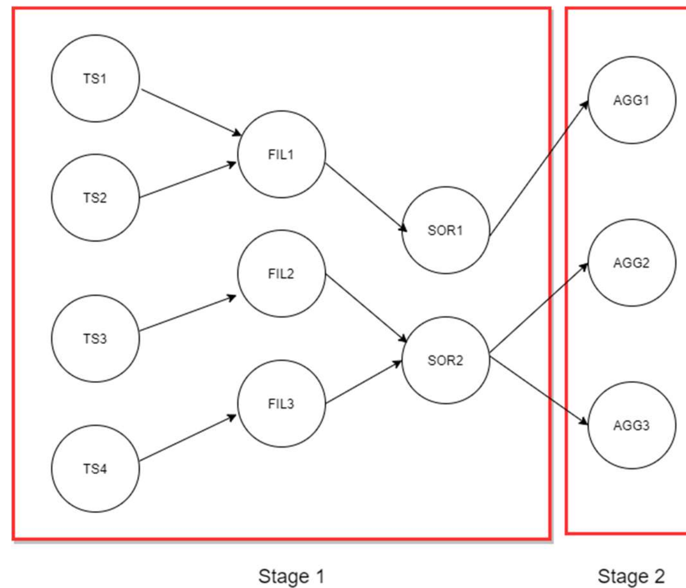


Figure 8. QEP is divided into different stages after being compiled from the query

Table 2. List of selected features

DIFF_SELECTIVITY(C_1)
DIFF_SELECTIVITY(C_2)
DIFF_SELECTIVITY(C_n)
DIFF_NDV(C_1)
DIFF_NDV(C_2)
DIFF_NDV(C_n)
DIFF_HISTOGRAM(C_1)
DIFF_HISTOGRAM(C_2)
DIFF_HISTOGRAM(C_n)

4.3.2 Machine Learning Model Selection

There exist a lot of machine learning models, but we need to choose a model that has high accuracy in predicting if a re-optimization is beneficial and incurs smaller overheads than the amounts of query execution time and monetary cost that it can save by avoiding unnecessary re-optimizations. The overheads incurred by a prediction model include the time to train the model (training time) and the time to apply the trained model for prediction (prediction time). For each database system, an individual model should be trained. In our case, as the model is trained offline, we are only concerned about the prediction time overhead. Applying different models trained by different learning algorithms may have different prediction time overheads. For example, applying a model created by a Neural

Network learning algorithm may have a different prediction time overhead compared with the prediction time overhead when applying a model trained by a Random Forest algorithm [33] as the former model is one tree while the latter model consists of multiple trees. This overhead may be different even when applying different models that are trained by the same learning algorithm. For example, checking a Neural Network with 50 layers to derive a prediction is far different from checking a Neural Network with 1000 layers

4.3.3 Applying Supervised Learning Model to Query Re-Optimization

In this section, we illustrate how the trained model is applied during the query execution, and the details are provided in Algorithm 1 in Figure 10. From Line 1 to Line 3, it initializes the OldStatistics, Result, and MergeTable with an empty value. The OldStatistics and the NewStatistics in the following are the regular data statistics used in existing database systems. Using those two variable names is to distinguish the data statistics before and after updating.

MergeTable is a temporary table in the memory. It is created when a query is received by the optimizer and destroyed after this entire query has been executed. The purpose of the MergeTable is to store the temporary results of any executed physical query operators. Line 4 uses the GenerateQEP function to generate an initial query execution plan from the query. From Figure 11, the GenerateQEP function first uses an existing logical plan generator to convert the query into a logical query execution plan (Line 1) and then uses an existing

query execution plan generator to convert the query into a physical query execution plan (Line 2). And Line 3, this physical query execution plan is merged with the merge table by Merge function.

Hash code of executed operator types	Executed Operator Result
B422ED….// TS(Suppliers)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA 3, 32, Oracle, 13, 5th street, SF, CA
F2AC13……// TS(Suppliers),FIL(sctiy=Seattle,sstate=WA)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA

Figure 9. Example of MergeTable

Figure 9 shows an example of MergeTable. There are two columns in the MergeTable. The first column stores the hash code of executed operator types, and the second column stores the result of those executed operators. Notice that, we record the hash code of executed operator types only, not the actual operators. The reason is that the purpose of using MergeTable is to find the reusable results of executed operators. For the physical operators of the same type, the result is the same so that all of those operators can be replaced with the same operator called "ReadMergeTable". For example, for reading table A, optional physical operators plan can be "FileScan (A)" in one physical query execution plan or "IndexScan (A)" in another plan. But, both physical operators, belong to the same operator type and generate the same results after execution. Hence, we only recode "TableScan" for both.

Besides that, there are two reasons for storing hash code here. First, using hash code can quickly locate the result of executed operators. Second, hash code can locate the exactly matched executed operators. Exactly matched operators mean not only those operators have to be the same type respectively, but also the order of the operators has to be the same as well. By using hashing can make sure both of the types and the order are matched correctly.

The Merge function explains how a physical query execution plan is merged with the MergeTable. Suppose we have 4 operators *Op1*, *Op2*, *Op3*, and *Op4* in the physical query execution plan and they will be executed respectively. From Line 1 to Line 4 shown in Figure 11, we initialize *PreparedOperators* to empty and two control variables *i* and *j* to 1. In Line 7, we put Op1 into the *PreparedOperators* and convert it to a hash code. Line8 checks if this hash code has any match in the MergeTable. In Line 9 and 10, If it is matched to one hash code in the 1st column of the merge table, this *Op1* will be replaced with *ReadMergeTable (hash code)*. So, in the modified physical query execution plan, *Op1* will not be executed, and instead, we just need to read the result of executing *Op1* from the MergeTable. Then the algorithm goes back to Line 7 again, both *Op1* and *Op2* are put in the *PreparedOperators* and are converted into hash code. This hash code again is checked for if there is any match in the MergeTable. If the match is found, the same process from Line 9 and Line 10 repeats. If there is no match that can be found, in Line 13, the *PreparedOperators* is set to empty. The algorithm goes back to Line 7, and at this time,

Op2 alone is put into the *PreparedOperators* and is converted into hash code to find any matches in the MergeTable.

If the match is found, *Op2* and *Op3* are converted and checked in the next step. And if the match is still found, *Op2*, *Op3*, and *Op4* are converted and checked. The Merge function terminates after the last operators in the physical query execution plan is converted to hash code and checked for a match. Simply speaking, this function starts checking the hash code of the first operator *Op1* in the physical query execution plan as the beginning. Then add the next operator and these new added operators are converted together with all the previous operators and checked for a match. When there is no match found in the MergeTable, it starts from the second operator *Op2* and repeats the same process. Whenever the final operator is converted and checked for a match, the whole function terminates. The following Table 3 gives an example of the order of those operators are converted and checked.

Table 3. Sequence of searching matching operators in MergeTable

Sequence	**Operators being converted and checked**	**If match is found**
1	*Op1*	Yes
2	*Op1, Op2*	Yes
3	*Op1, Op2, Op3*	No
4	*Op2*	Yes
5	*Op2, Op3*	No

6	*Op3*	Yes
7	*Op3, Op4*	No
8	*Op4*	No
9	Terminates as no more operators are found in the physical query execution plan	

Here, we continue from the main function in Figure 10.

In Line 5, The first operator or first stage of operators are executed and in Line 6, the MergeTable is updated. The 1st column is updated by converting the executed operator in Line 5 into hash code and the 2nd column is updated by the results of executing this operator.

In Line 7 and Line 8, we update the data statistics and compute the difference between OldStatistics and NewStatistics. In Line 9, when there still exists an operator that has not been executed. In Line 10, we use the decisional model to decide if the query needs to be re-optimized. If the answer is "YES" in Line 11, we will use the same function GenerateQEP again to generate a new QEP in Line 12. From Line 13 to Line 14, we execute the next available operator or next stage of operators in the new QEP and update the MergeTable with the results generated in Line 13. In Line 15, we set the current QEP to the new QEP. In Line 16, if the re-optimization is "No", the QEP is not re-optimized, and the next available operator or next stage of operators are executed continuously in Line 17. From Line 19 to Line 22, no matter the QEP is re-optimized or not, the data statistics are always updated and prepared for the decision model to make a decision on re-optimization. In Line 25, after all

the operators are executed, the loop from Line 9 to Line 24 terminates and the final results are returned to the user.

Algorithm 1: Query Processing with Machine Learning-based Re-Optimization (ReOptML)

INPUT: Query // SQL query
OUTPUT: The query result set of the input query
1. OldStatistics = get current data statistics
2. Result = Ø
3. MergeTable = Ø
   //query optimizer generates a physical query execution plan
4. QEP = GenerateQEP (OldStatistics, Result, Query, MergeTable)
5. Result = execute the next available operator or stage if stage is available in QEP
   // record the hash codes and results of the executed operators
6. MergeTable = UpdateMergeTable (Result)
   // call query optimizer to update the data statistics
7. NewStatistics = UpdateDataStatistics ()
8. DiffStatistics = compute difference between OldStatistics and NewStatistics
   // if there still exists an operator that has not been executed
9. **while** QEP ≠ Ø
   // using the learning model to predict whether the query should be re-optimized
10. ReOptDecision = RunPredictiveModel (DiffStatistics)
11. **if** ReOptDecision = 'YES'
    //query optimizer generates a new physical query execution plan
12. NewQEP = GenerateQEP (NewStatistics, Result, Query, MergeTable)
13. Result = execute the next available operator or stage if stage is available in NewQEP
    //record the hash codes and results of the executed operators
14. MergeTable = UpdateMergeTable (Result)
15. QEP = NewQEP
16. **else if** ReOptDecision = 'NO'
17. Result = execute the next available operator or stage if stage is available in QEP
18. **end if**
19. **if** QEP ≠ Ø
20. OldStatistics = NewStatistics
    // call query optimizer to update the data statistics
21. NewStatistics = UpdateDataStatistics ()
22. DiffStatistics = compute difference of OldStatistics and NewStatistics
23. **end If**
24. **end while**
25. **return** Result

Figure 10. Query processing algorithm with machine learning-based re-optimization

```
Function GenerateQEP (Statistics, Result, Query, MergeTable)
1. LogicalPlan = LogicalPlanGenerator (Query, Statistics)
2. PhysicalPlan = PhysicalPlanGenerator (LogicalPlan, Statistics)
3. ModifiedPhysicalPlan = Merge (PhysicalPlan, MergeTable)
4. return ModifiedPhysicalPlan

Function Merge (PhysicalPlan, MergeTable)
1.  PreparedOperators = {}
2.  ModifiedPhysicalPlan = PhysicalPlan
3.  i = 1
4.  j = 1
5.  while there exists one operator has not been visited
6.     i = i + 1
7.     PreparedOperators. Add (Op_j)
       // convert the prepared operators to hashcode
8.     HashCode = Hash (PreparedOperators)
9.     if MergeTable.found (HashCode)
         //Replace the executed operators with one operator called "ReadMergeTable"
         //which reads the results of those executed operators from the MergeTable
10.      ModifiedPhysicalPlan.Replace (PreparedOperators, Read MergeTable (HashCode)
11.      j = j + 1
12.    else
13.      PreparedOperators = {}
14.      j = i - 1
15. end while
16. return ModifiedPhysicalPlan
```

Figure 11. Merge and GenerateQEP function

4.4 Summary

In this chapter, we presented ReOptML, an algorithm that uses supervised machine learning to re-optimize queries in a cloud DBMS. In this algorithm, a well-trained supervised machine learning model takes the difference of data statistics before and after executing a portion of a QEP as input to predict whether the re-optimization is beneficial or not. Only beneficial re-optimizations are then triggered. In the next chapter, we introduce our third and fourth proposed algorithms, ReOptRL and SLAReOptRL, which

conduct query re-optimization considering query response time, monetary costs, and SLA requirements. The algorithms do not depend on labeled training data and updated data statistics.

CHAPTER V
PROPOSED REINFORCEMENT LEARNING BASED QUERY RE-OPTIMIZATION ALGORITHMS FOR CLOUD DATABASE SYSTEMS (ReOptRL and SLAReOptRL)

In cloud database systems, a Service Level Agreement (SLA) is signed between users and cloud providers before any service is provided. If an SLA is violated, cloud providers will need to pay a penalty [37]. Thus, from the profit-oriented perspective for the cloud providers, query re-optimization is multi-objective optimization that minimizes not only query execution time and monetary cost but also SLA violation. However, none of the existing query re-optimization algorithms consider all three objectives. To fill this gap, in this chapter, we introduce reinforcement learning based query re-optimization algorithms for cloud database systems, ReOptRL and SLAReOptRL, two novel query re-optimization algorithms for cloud database systems based on deep reinforcement learning. ReOptRL considers query execution time and monetary costs. It bootstraps a QEP generated by an existing query optimizer and dynamically changes the QEP during the query execution based on the optimization model which keeps learning from incoming queries. The QEP is adjusted based on the recent performance of the same query so that the algorithm does not rely on cost estimations. SLAReOptRL extends ReOptRL by also including SLA requirements in the adjustment of QEPs.

We present ReOptRL and SLAReOpt in the following sections of this chapter. In Section 5.1, the reinforcement learning algorithm is briefly introduced. In Section 5.2, we give the

motivations for designing ReOptRL. In Section 5.3 and Section 5.4, the overview and details of ReOptRL are given. In Section 5.5, the design of the reward function is introduced. In Section 5.6, we describe how we extend ReOptRL to SLAReOptRL.

5.1 Reinforcement Learning-Based Algorithms for Query Re-Optimization

As described in [38] and shown in Figure 12, reinforcement learning describes the interaction between an agent and an environment. The possible actions that the agent can take given a state S_t of the environment are denoted as $A_t = \{a_0, a_1, \ldots, a_n\}$. The agent acts as the action set A_t based on the current state S_t of the environment. For each action taken by the agent, the environment gives a reward rt to the agent and the environment turns into a new state S_{t+1}, and the new action set is A_{t+1}. This process repeats until the terminal state is reached. These steps form an episode. The agent tries to maximize the reward and will adjust after each episode. This is known as the learning process.

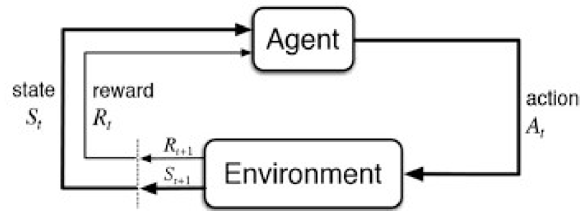


Figure 12. General procedure of reinforcement learning [38]

5.2 Motivation of ReOptRL

Traditionally, the query optimizer evaluates the time and monetary costs of different QEPs to derive the best QEP for a query before execution. These time and monetary costs are

estimated based on the data statistics available to the query optimizer at the moment when the query optimization is performed. These statistics are often approximate, which may result in inaccurate estimates for the time and monetary costs needed to execute the query. Thus, the QEP generated before query execution may not be the best one.

To solve the problem, researchers have developed learning-based algorithms to adjust the data statistics to get more accurate cost estimations [31]. These methods are heuristic-based and the adjustment of QEP is not adaptable to a dynamic environment. Later, machine learning-based algorithms are introduced [39, 28]. More accurate cost estimations are made by data statistics estimated by machine learning models. The optimizer uses these cost estimations to adjust the QEP. More recently, the work in [39] presents a machine learning-based approach to learn cardinality models from previous job executions, and these models are then used to predict the cardinalities in future jobs. Again, even those methods improve the accuracy of data statistics estimation such as cardinalities, the overall performance is not improved much. This is usually because updating data statistics for the optimizer to use is a very expensive operation by itself. This becomes the main source of negative impacts on the overall performance. In work [40], the authors examine the use of deep learning techniques in database research. With supervised machine learning, labeled data must be available in advance for training, which is not always possible to obtain. To avoid this problem, reinforcement learning (RL) is used. Some algorithms used reinforcement learning in adjusting their QEPs also. However, these adjustments are only focusing on the join order

of queries [6]. None of the reviewed algorithms addresses monetary costs and SLA requirements for cloud databases.

There are various kinds of RL algorithms that have been proposed. Q-Learning is one of the popular value-based RL algorithms [41]. In Q-Learning, a table (called Q-table) is used to store all the potential state-action pairs (S_n, a_n) and an evaluated Q-value associated with this pair. When the agent needs to decide which action to perform, it looks up the Q-value from the Q-table for each potential action under the current state and selects and performs the action with the highest Q-value. After the selected action is performed, a reward is given, and the Q-value is updated using the Bellman equation [38]:

$$Q(S_t, a_t) \leftarrow Q(S_t, a_t) + \alpha\big(R_t + \Upsilon\, Q(S_{t+1}, a_{t+1}) - Q(S_t, a_t)\big) \quad (3)$$

In Equation (3), $Q(S_t, a_t)$ is an evaluated value (called Q-value) for executing Action a_t at State S_t. This value is used to select the best Action to perform under the current state. To keep this value updated with accurate evaluation is the key to reinforcement learning. α is the learning rate and Υ is the discount rate. These two values are constant between 0 and 1. The learning rate α controls how fast the new Q-value is updated. The discount rate Υ controls the weight of future rewards. If $\Upsilon = 0$, the agent only cares for the first reward, and if $\Upsilon = 1$, the agent cares for all the rewards in the future [38]. R_t is the reward; the detailed reward function in this algorithm is described in Section 5.5.

5.3 Overview of ReOptRL

In this chapter, how a deep reinforcement learning algorithm is used in query processing to select the best action for the performance of queries is introduced. Two algorithms are presented. In this section, we present the first algorithm which is a non-SLA-based algorithm (ReOptRL). In this algorithm, a query will be converted into a logical plan by a traditional query parser. Then for each logical operator, we use a deep reinforcement learning model to select the exact physical operator and machine to execute the logical operator so that each operator execution is optimized in order to gain the maximum improvement on the overall performance. These machines are called containers in the rest of this chapter. These selections learn from the same operator executed in the system previously. As in large applications, there will be a large number of queries running at the same time. It is reasonable to refer to the performance of the same operator in the system because the times of the previous executions of the same operator are very close to each other. The second algorithm that we present is the SLA-based query re-optimization algorithm (SLAReOptRL). The detail of this algorithm will be presented in Section 5.6.2.

Notice that, in the scenario of this chapter, queries are processed in a cloud database system. There is a large number of available containers on which a single query operator can be executed. There are potentially many state-action pairs in the Q-table. Iterating a large Q-table incurs extra time overhead which delays the query execution. To solve this issue, Deep Q Network (DQN) [42] is applied as reinforcement learning for query re-optimization. DQN

works similarly to Q-Learning. The major difference is that, as shown in Figure 13, given a state, instead of using the Q-table, it uses a neural network to estimate the Q-values for all the potential actions. After each action is performed, a reward is given, and the Q-value is updated using the Bellman equation. This updated Q-value is then used to adjust the weights of the neural network using the back-propagation method. As the Q-values of all the actions are evaluated at once, there is no need to look up the Q-value from the Q-table for each action repeatedly. Thus, the processing time of running the DQN method is much shorter than that of Q-Learning. Since query response time is critical and to reduce the time overhead, we apply DQN to our query re-optimization algorithm. Figure 13 describes the different procedures of Q-Learning and DQN.

5.4 Details of ReOptRL

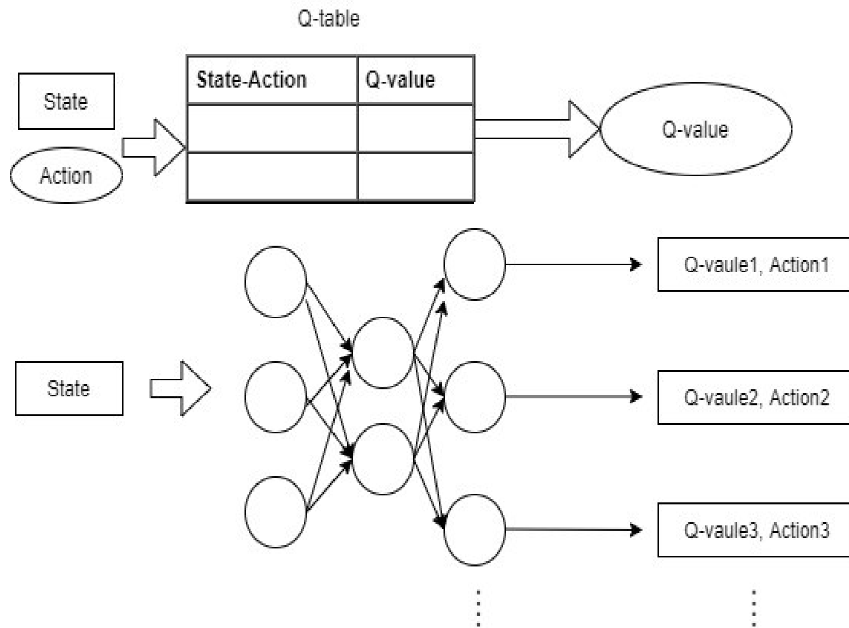


Figure 13. Procedures of Q-Learning (the top figure) and DQN (the bottom figure)

In this algorithm, the policy gradient deep RL algorithm [41] is used for query re-optimization. This algorithm uses a deep neural network to help the agent decide the best action to perform under each state. In this work, the agent is the query optimizer, an action is a combination of a physical operator to execute a logical operator and a machine to execute this operator, and a state is a fixed-length vector encoded from the logical query execution plan produced by a conventional query optimizer.

The input of the neural network is the current state. The input is sent to the first hidden layer of the neural network whose output is then sent to the second layer, and so on until the final layer is reached, and then an action is chosen. The policy gradient is updated using a sample of the previous episodes, which is an operator execution in our case. Once an episode is completed (which means a physical operator and a container to execute the physical operator are selected in our case), the execution performance is recorded, and a reward is received where a reward is a function to evaluate the selected action. The details of the reward function are explained later Section 5.5. The weights of the neural network are updated after several episodes using existing techniques, such as back-propagation [41].

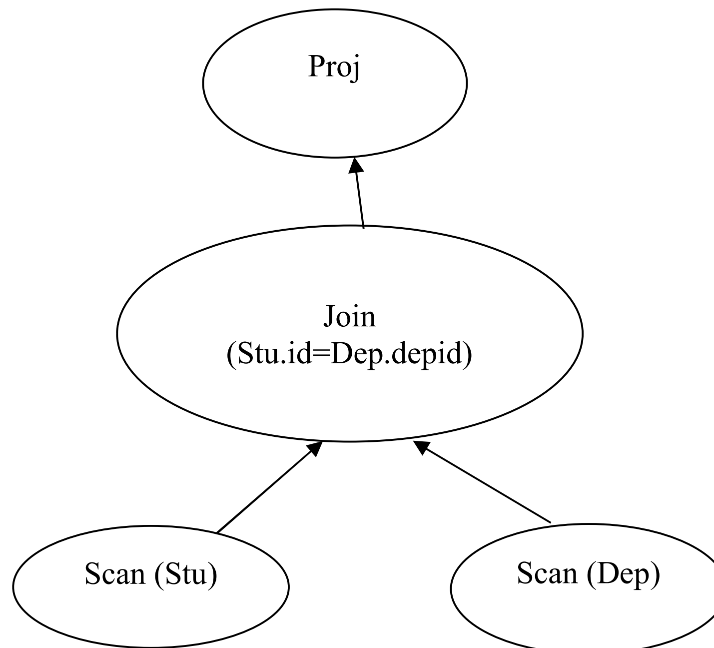
For the current QEP to represent the current state and to be used as the input of the neural network, we use a one-hot vector adapted from the recent work [28] to represent a QEP. Each component in a one-hot vector is mapped to an attribute in a relation. A component

has a value of 1 if the corresponding attribute is present in the query operator and 0 otherwise.

For example, we have the following SQL query:

*Select **
From Stu, Dep
Where Stu.depid=Dep.depid

This SQL query is first optimized by a conventional query optimizer which produces the following QEP:



Assume the schemas or the two tables are Stu (id, name, depid) and Dep (depid, name). The JOIN operator in this QEP can be represented as a one-hot vector V as follows:

V=[OperatorName, Stu.id, Stu.name, Stu.depID, Dep.depid, Dep.name …]

=[‘JOIN’, 0, 0, 1, 1, 0…]

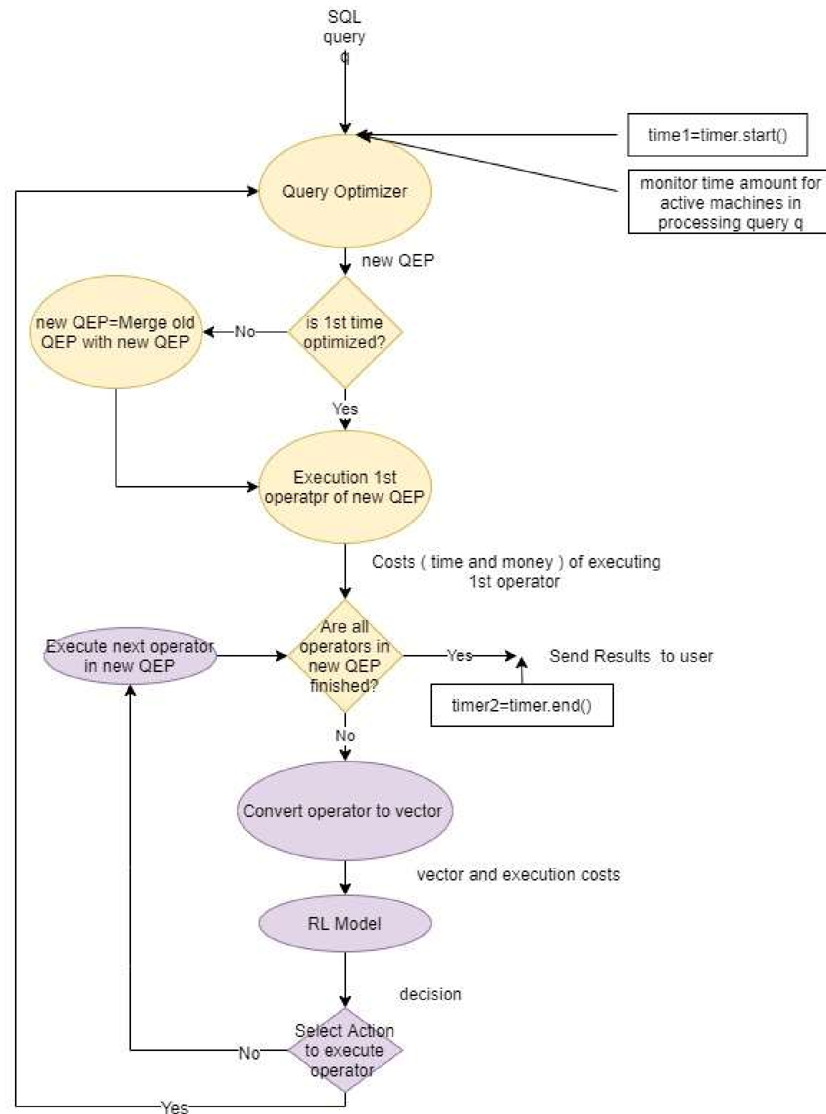


Figure 14. Procedure of ReOptRL

Figure 14 shows the major steps in query processing when ReOptRL is incorporated for query re-optimization. Firstly, the optimizer receives a query and then the query is compiled into a QEP. Secondly, the first available operator is converted into a vector representation and is sent to the RL model. The RL model will select the optimal action, which is the

combination of a selected physical operator and a selected container to execute the selected physical operator. The physical operators are generated by the query optimizer and the containers are those available on the cloud platform. Then the selected physical operator is executed, and the execution time and monetary costs of this execution are recorded to update the reward. Once the reward function is updated, the weights of the RL model are adjusted according to the updated reward. Then the updated RL model is ready for future action selections of the same operator. Figure 15 shows the pseudo-code of the proposed algorithm. First, a query is submitted to the query optimizer which generates the QEP for the query (Line 4). Then the QEP is converted into a one-hot vector representation (Line 7). This vector is sent to the RL model, which is a neural network as described in Section 5.1. The RL model will evaluate the Q-values for all the potential actions to execute the next available query operator (Line 8). Each of these actions consists of two parts, a physical operator, and a container to execute the physical operator. Then the action with the best Q-value will be selected and performed by the DBMS (Line 9). After that, the executed query operator is discarded from the QEP (Line 10). The reward is updated with the time and monetary cost needed to execute the operator and then the expected Q-value is updated by the Bellman Equation (3) with the updated reward (Lines 11-13). The weights of the neural network are updated accordingly by the back-propagation method (Line 14). This process repeats for each operator in the QEP and terminates when all the operators in the QEP are executed. The query results are then sent to the user (Line 17).

Algorithm: Query Processing with Reinforcement Learning Based Re-Optimization (ReOptRL)

INPUT: SQL query, Weight Profile wp, Reward Function R (),
Learning rate α, Discount rate ϒ
OUTPUT: The query result set of the input query

1. t = 0
2. Result = Ø
3. $Q_t = 0$
4. QEP = QueryOptimizer (query)
5. **while** QEP ≠ Ø
6. Op = next available operator in QEP
7. State S_t = convert QEP to a state vector
8. $Action_t$ = RunLearningModel (S_t, wp)
9. Result = Result ∪ Execute (Op, $Action_t$)
10. QEP = QEP - Op
11. Update R_t = R (wp, $Action_t$.time, $Action_t$.money)
12. Obtain Q-value of next state Q_{t+1} from the neural network
13. Update Q-value of current state Q_t = Bellman (Q_t, Q_{t+1}, R_t, α, ϒ)
14. Update Weights in the neural network
15. t = t + 1
16. **end while**
17. **return** Result

Figure 15. Query processing using reinforcement learning-based re-optimization

5.5 Reward Function

In ReOptRL, after an action is performed, the reward function is used to evaluate the action. This gives feedback on how the selected action performs to the learning model. The performed action with a high reward will be more likely to be selected again under the same state. The reward function plays a key role in the entire algorithm. According to the Bellman equation, if the reward of performing the previous action a_{t-1} is high on the state s_{t-1}, the Q-value will also be high. This means, given the same state, the action with the good previous performance will have a higher chance to be selected. In our algorithm, we would like the

actions with low query execution time and monetary cost to be the ones that will be more likely to be chosen. To reflect this feature, here we define the reward function as follows:

$$Reward\ R = \frac{1}{1+(W_t * T_{op}^q)+(W_m * M_{op}^q)} \qquad (4)$$

where W_t and W_m are the time and monetary weights provided by the user, and T_{op}^q and M_{op}^q are the time and monetary costs for executing the current operator *op* in the query q.

According to this reward function, the query is executed based on the user's preference which is either the user wanting to spend more money for a better query execution time or vice versa. We call these preferences *Weights*. These weights defined by the user are called *Weight Profile* (*wp*), which is a two-dimensional vector, and each dimension is a number between 0.0 to 1.0. Notice that, the user only needs to specify one dimension of the weight profile, the other dimension is computed with *1-Weight* automatically. For example, if a user demands fast query response time and is willing to invest more money to achieve it, a possible weight profile for this user could be $<W_t=0.9, W_m=0.1>$. The detail can be found in Section 4.2.

This reward function is a monotonic decreasing function. With the increase of $(W_t * T_{op}^q) + (W_m * M_{op}^q)$, which is the total costs for executing a query operator, the reward decreases. Notice that, as $(W_t * T_{op}^q) + (W_m * M_{op}^q)$ approaches zero, the reward approaches positive infinity. When this situation happens, if an action *A* is performed with small total costs, then

A will always be selected and performed, and all the other actions will be ignored. This is not desirable, and to keep the relationship of reward and total costs close to linear, we use 1+$(W_t * T_{op}^q) + (W_m * M_{op}^q)$ as the denominator in the reward function. In summary, if performing an action takes high costs, this action will be less likely to be chosen in the future. Also, according to the following Equation (5), if an action is selected but fails to perform due to some error which results in the time cost that becomes positive infinity, the reward is 0. This can make such an action less likely to be chosen again in the future. The failure of an action to perform can be caused by different reasons, such as the wrong physical query operator being chosen, or the container assigned being unavailable.

$$\lim_{T_{op}^q \to \infty} \frac{1}{1+\left(W_t * T_{op}^q\right)+\left(W_m * M_{op}^q\right)} = 0 \qquad (5)$$

5.6 SLA-Aware Reinforcement Learning-based Algorithms for Query Re-Optimization (SLAReOptRL)

In this section, we introduce the algorithm, SLAReOptRL. In Section 5.6.1, we explain what SLA is and in Section 5.6.2, we show how the SLA is used in query re-optimization by SLAReOptRL.

5.6.1 SLA Definition

SLA is a contract between cloud service providers and consumers, mandating specific numerical target values which the service needs to achieve. Considering SLA in query processing is important in cloud databases. Optimization of QEPs should be done in such a

way that will not violate the SLA requirements, while considering other objectives, such as query execution time and monetary costs; otherwise, if the SLA violation happens, the cloud service providers need to pay a penalty to their users in a form such as money or CPU credits [37]. From a profit-oriented perspective, cloud service providers would want to keep the SLA violations as low as possible. Different cloud service providers implement different SLAs with their users. Many commercial cloud systems use "server availability" as their SLA requirement. This means if a server fails, the cloud providers will pay a certain number of credits to their users. In recent research, using time and monetary costs to execute a query as the SLA requirements have been studied [40]. We find them practical and more specific to users and, thus, adopt the same SLA requirements in this algorithm.

5.6.2 Extending ReOptRL to Consider SLA Violation

Our proposed algorithm, SLAReOptR, extends the ReOpRL algorithm presented in Section 5.4 to also consider SLA requirements besides query execution time and monetary costs. In particular, it extends the reward function as defined in Equation (6) to make it possible to select the best action according to the SLA requirements.

$$Reward\ R = \frac{1}{1+(W_t*\left(T_{op}^{q}+P_t\right))+(W_m*(M_{op}^{q}+P_m))} \qquad (6)$$

where T_{op}^{q} and M_{op}^{q} are the time and monetary costs for executing the current operator *op* in the query q.

$$P_t = \alpha_{op} * delay_time,\ \ P_m = \alpha_{op} * exceeded_money \qquad (7)$$

where α_{op} is the operator impact rate of the operator type *op*.

$$delay_time = \begin{cases} 0 \\ T_{op}^{q} - SLA.T_{op}^{q} \ \ if \ T_{op}^{q} > SLA.T_{op}^{q} \end{cases} \quad (8)$$

$$exceeded_money = \begin{cases} 0 \\ M_{op}^{q} - SLA.M_{op}^{q} \ \ if \ M_{op}^{q} > SLA.M_{op}^{q} \end{cases} \quad (9)$$

In this reward function (Equation 6), P_t and P_m as defined in Equation 7 reflect the extra costs for executing a query operator if the SLA is violated. If the SLA is not violated for executing every operator, then this equation is the same as the reward function used in ReOptRL (Equation 4). In Equations (8) and (9), $delay_time$ is the amount of difference between the actual time to execute a query operator and the maximum time allowed to execute this query operator as specified in the SLA. The same idea applies to $exceeded_money$ for monetary costs. We use these two values to quantify the amount of SLA violations on query execution time and monetary cost. In Equation (7), these two values are used to compute P_t and P_m. It shows that the larger the amount of SLA violations, the smaller the reward becomes. We build the reward function this way so that the reward is related to the amount of SLA violations.

Also, we use the query operator impact rate α_{op} to scale up the impact of SLA violations on different types of operators. For example, the impact of the JOIN operator is usually larger than the impact of other types of operators. Notice that, the SLA requirements presented in Equations (8) and (9) are not the same as the SLA requirements specified in

the agreement. While the time and monetary costs are defined as "amount per query" in the SLA requirements specified in the agreement, they are defined as "amount per query operator" in Equations (8) and (9). Here, we average the SLA requirements specified in the agreement by the total number of operators in the QEP. Besides this simple method of computing SLA requirements, we plan to study alternative ways in our future research.

5.7 Summary

In this chapter, we presented ReOptRL, an algorithm that uses reinforcement learning to re-optimize queries in a cloud DBMS. In this algorithm, instead of using an existing query optimizer repeatedly in re-optimization, for a given query, once the query optimizer produces the QEP for the query, the algorithm uses a reinforcement learning model to select the best action to execute the next available operator in the QEP. The time and monetary costs of executing this operator are used as a reward to improve the accuracy of the learning model. Then, we presented SLAReOptRL, an extended version of ReOptRL to reduce SLA violation where the reward function supports the selection of actions that meet SLA requirements. In the next chapter, the theoretical analysis and experimental results of all the four proposed algorithms are presented.

CHAPTER VI
PERFORMANCE ANALYSIS

In this chapter, we present the performance analysis of the proposed algorithms, ReOpt, ReOptML, ReOptRL and SLAReOptRL. In Section 6.1, we present the performance analysis theoretically. We analyze the time complexity and provide proof of the correctness of these algorithms. In Section 6.2, we present the performance analysis experimentally. Comprehensive experiments are conducted on each of the algorithms and the results are compared to the results of the state-of-the-art algorithms.

6.1 Theoretical Analysis

In this section, first, in Section 6.1.1, we provide the proof of correctness of the three algorithms, ReOpt, ReOptML, and ReOptRL. We prove that the query results are correct after a query is processed by these three algorithms. In Section 6.1.2, we provide the time complexity analysis of these three algorithms.

6.1.1 Proof of Correctness of ReOpt, ReOptML and ReOptRL

6.1.1.1 Proof of Correctness of ReOpt and ReOptML

As ReOpt and ReOptML use the same method to do the re-optimization. The difference between them is that in ReOptML, the re-optimization only happens when the decision model says "Yes" while in ReOpt, the re-optimization always happens when an operator or a stage of operators finishes execution. From the correctness perspective, they can share the same proof. Thus, in this section, we focus on analyzing the ReOptML algorithm

theoretically. Figure 10 and Figure 11 in Section 4.3.3 show the details of ReOptML. Here, to prove the correctness of ReOptML, we show theoretically that after a QEP is re-optimized and merged with the MergeTable, the results of executing the new QEP do not change.

Definition:

Let the Ordered Sequence,

$P_1 = (O_1, O_2, \ldots, O_n)$ denotes a physical query execution plan generated by a query optimizer. Similarly, $P_2 = (R_1, R_2, \ldots, R_m)$ denotes a physical query execution plan generated by a query optimizer. Also, we have $P_k = (O_x, O_{x+1}, \ldots, O_{x+a})$, $1 \le x \le n-a$ denotes a sub physical query of P_1 and similarly, $P_q = (R_y, R_{y+1}, \ldots, R_{y+b})$, $1 \le y \le m-b$ denotes a sub physical query of P_2.

$Exe\ (O_1, O_2, \ldots, O_n) = Exe(O_1) \Rightarrow Exe\ (O_2) \ldots \Rightarrow Exe\ (O_n)$ denotes that the results of executing operators $O_1, O_2, \ldots, O_n$ is the same as executing operator O_1 first and then using the results to execute the operator O_2. then using the results to execute O_3, and so on, until O_n is executed.

Theorem 1:

Given two physical query execution plans P_1 and P_2 from the same query by the same query optimizer,

If $Exe\ (P_q) = Exe\ (P_k)$,

Then $Exe\ (P_1) = Exe\ (R_1, R_{y+1}, \ldots, R_{y-1}) \Rightarrow Exe\ (P_k) \Rightarrow Exe\ (R_{y+b}, R_{y+b+2}, \ldots, R_m)$

In Theorem 1, P_1 is the QEP before re-optimization and being merged. P_2 is the QEP after re-optimization and being merged. P_k contains the operators that have been executed. The operators in P_k are converted into a hash code and stored in the 1st column of the MergeTable and the results of executing P_k are stored in the 2nd column in the MergeTable. P_q contains the operators in P_2 that are being replaced with the ReadMergeTable.

Proof:

Proof by Induction

Step 1

Basic Case,

For n=1 and m=1

We have Exe (P_k) = Exe (P_1) and Exe (P_q)= Exe(P_2)

Right Hand Side: Exe (R_1, R_{y+1}, …, R_{y-1}) ⇒ Exe (P_k) ⇒ Exe (R_{y+b}, R_{y+b+2}, …, R_m) = Exe(P_k),

Left Hand Side: Exe(P_1) = Exe (P_k),

Then Exe (P_1) = Exe (R_1, R_{y+1}, …, O_{y-1}) ⇒ Exe (P_k) ⇒ Exe (R_{y+b}, R_{y+b+2}, …, R_m)

Step 2

Assume for some n and m, the theorem is true.

If Exe (P_k) = Exe (P_q)

then Exe(P_1) = Exe (R_1, R_{y+1}, …, R_{y-1}) ⇒ Exe (P_k) ⇒ Exe (R_{y+b}, R_{y+b+2}, …, R_m)

Step 3

Show for some n+1 and m+1, the theorem is true.

$P'_1 = (O_1, O_2, \dots, O_n, O_{n+1})$, $P'_2 = (R_1, R_2, \dots, R_m, R_{m+1})$, $P'_k = (O_x, O_{x+1}, \dots, O_{x+a+1})$,

$P'_q = (R_y, R_{y+1}, \dots, R_{y+b+1})$

If $Exe(P'_q) = Exe(P'_k)$

Then $Exe(P'_1) = Exe(R_1, R_2, \dots, R_{y-1}) \Rightarrow Exe(P'_k) \Rightarrow Exe(R_{y+b+1}, R_{y+b+2}, \dots, R_m, R_{m+1})$

As P'_1 and P'_2 are generated from the same query by the same optimizer.

$Exe(P'_1) = Exe(P'_2)$

$= Exe(P_2) \Rightarrow Exe(R_{m+1})$

$= Exe(R_1, R_2, \dots, R_{y-1}) \Rightarrow Exe(R_y, R_{y+1}, \dots, R_{y+b}) \Rightarrow Exe(R_{y+b+1}, R_{y+b+2}, \dots, R_m) \Rightarrow Exe(R_{m+1})$

$= Exe(R_1, R_2, \dots, R_{y-1}) \Rightarrow Exe(R_y, R_{y+1}, \dots, R_{y+b}, R_{y+b+1}) \Rightarrow Exe(R_{y+b+2}, R_{y+b+3}, \dots, R_m) \Rightarrow Exe(R_{m+1})$

$= Exe(R_1, R_{y+1}, \dots, R_{y-1}) \Rightarrow Exe(P'_q) \Rightarrow Exe(R_{y+b+2}, R_{y+b+3}, \dots, R_m) \Rightarrow Exe(R_{m+1})$

$= Exe(R_1, R_{y+1}, \dots, R_{y-1}) \Rightarrow Exe(P'_k) \Rightarrow Exe(R_{y+b+2}, R_{y+b+3}, \dots, R_m, R_{m+1})$

This proof shows that, after a query execution plan is re-optimized (from P_1 to P_2) and being merged with MergeTable (from P_k to P_q), the result of executing this query execution plan does not change. We use an example to illustrate how a query is re-optimized using ReOptML and also show its connection to the Theorem 1.

Suppose we have the following query,

SQL:

Select sname

From Suppliers, Supplies

Where Suppliers.sno=Supplies.sno

And Suppliers.scity="Seattle" And Suppliers.sstate="WA" And Supplies.pno=2

This query is converted to the following logical plan (represented in relational algebra) using GenerateQEP, Line 1

$$\pi_{sname}(\sigma_{scity=Seatt \quad \wedge sstate=WA}(Suppliers) \bowtie \sigma_{pno=2}(Supplies))$$

After that, the above logical plan is converted to the following physical query execution plan (represented in the graph) using GenerateQEP, Line 2

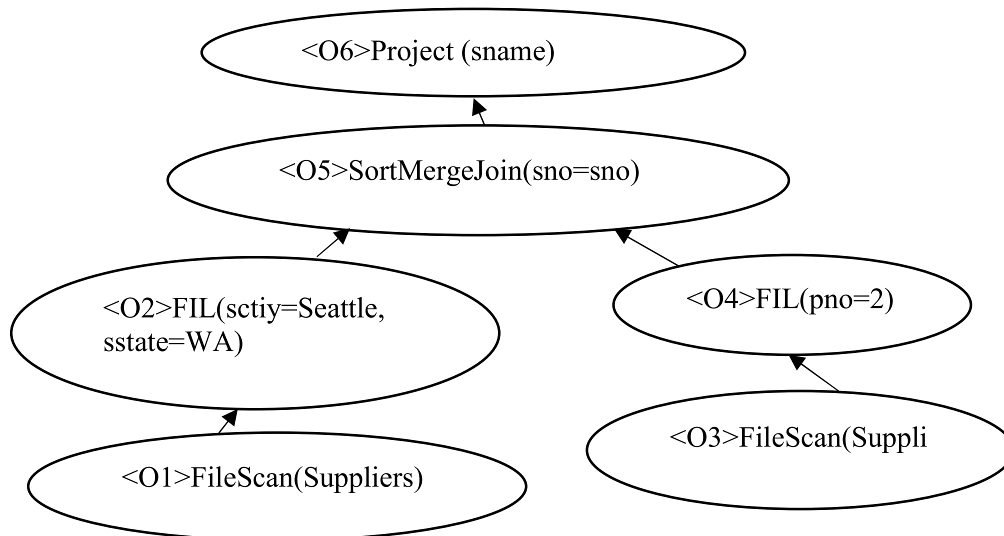


Figure 16. QEP P_1 generated by the query optimizer before re-optimization

$P_1 = (O_1, O_2, O_3, O_4, O_5, O_6)$

Step 1

O_1 is sent to be executed, and the re-optimization decision is “No”.

After O_1 is executed, the MergeTable is updated.

MergeTable

Hash Code of Executed Operator Type	Executed Operator Result
B422ED….// TS(Suppliers)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA 3, 32, Oracle, 13, 5th street, SF, CA

Then O_2 is sent for execution, and the re-optimization decision is “Yes”

After O_2 is executed, the MergeTable is updated as follows:

Hash Code of Executed Operator Type	Executed Operator Result
B422ED….// TS(Suppliers)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA 3, 32, Oracle, 13, 5th street, SF, CA
F2AC13……// TS(Suppliers),FIL(sctiy=Seattle,sstate=WA)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA

Suppose after re-optimization, we have the same QEP this time.

$P_2 = (R_1, R_2, R_3, R_4, R_5, R_6)$

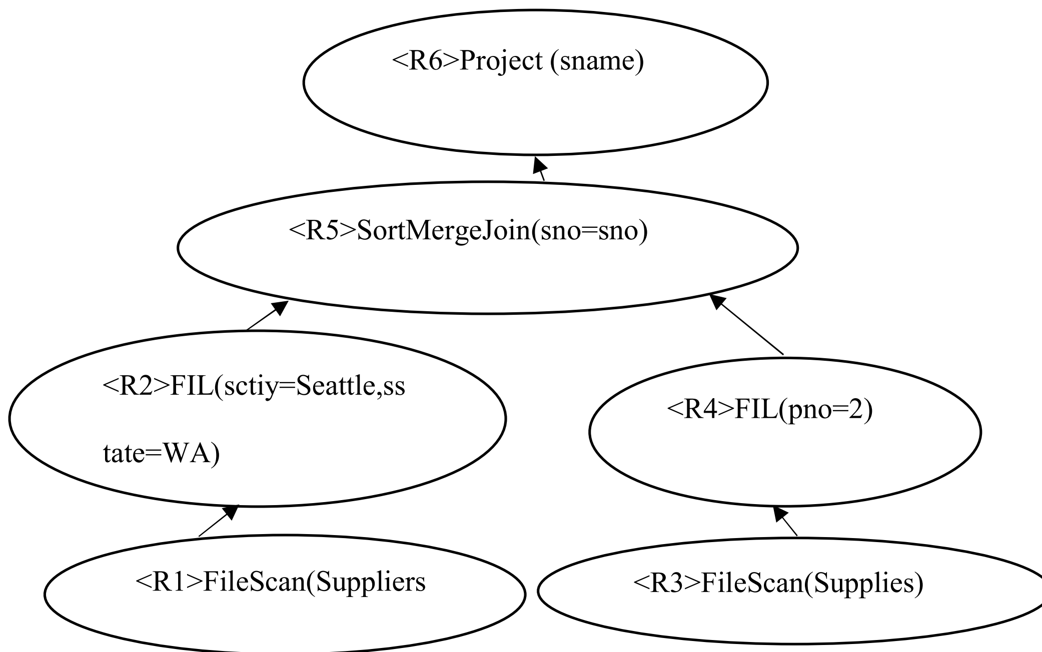


Figure 17. QEP P_2 after 1st re-optimization

We find O_1 in the MergeTable. Merge P_2 and MergeTable

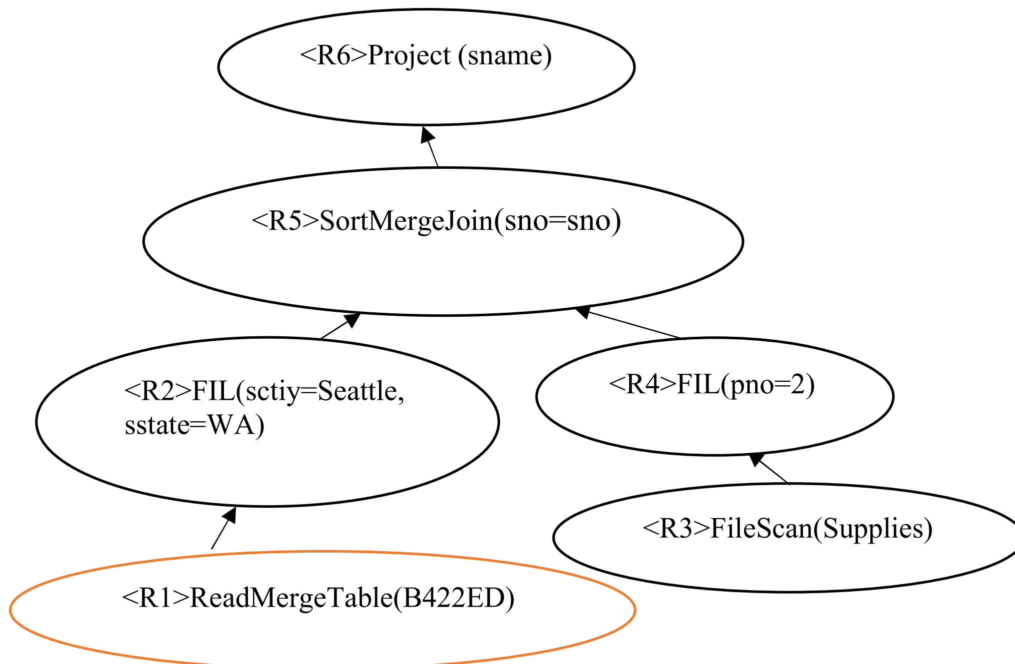


Figure 18. QEP P_2 after operator O_1 is merged

We find O_1, O_2 in the MergeTable. Merge P_2 and MergeTable

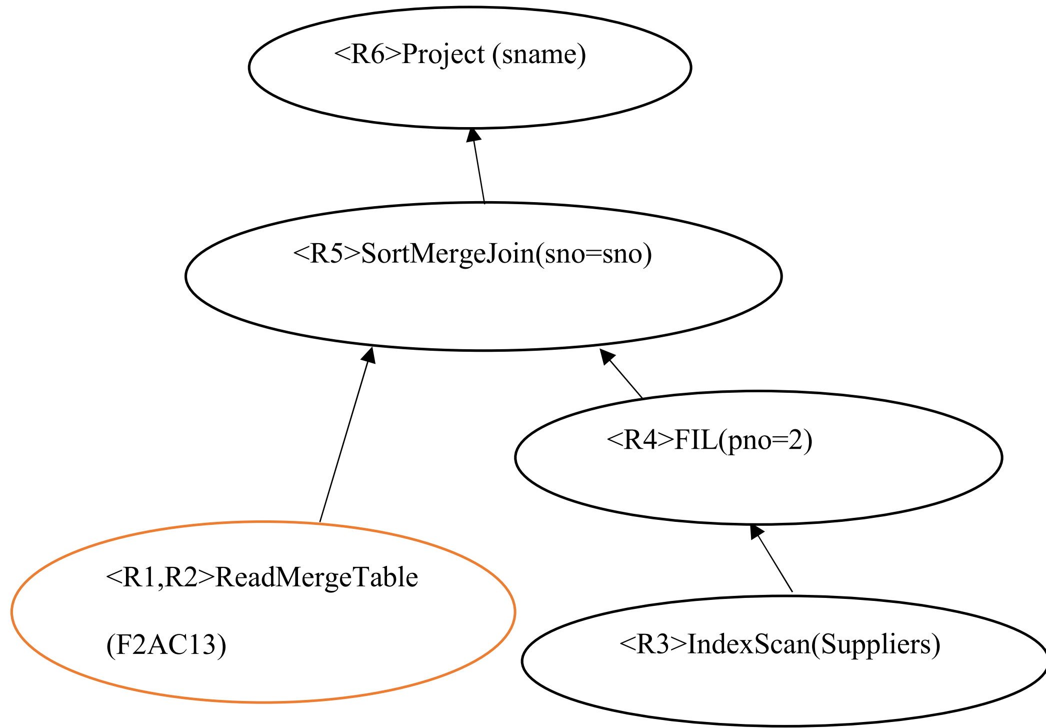


Figure 19. QEP P_2 after operators O_1 and O_2 are merged

Step 2

Since we cannot find any more hash code matches in the MergeTable, the P_2 is the above.

At this point, $P_1 = (O_1, O_2, O_3, O_4, O_5, O_6)$, $P_2 = (R_1, R_2, R_3, R_4, R_5, R_6)$, $P_k = (O_1, O_2)$, $P_q = (R_1, R_2)$

$Exe(P_2) = Exe\ (R_1, R_2, R_3, R_4, R_5, R_6) = Exe\ (R_1, R_2) \Rightarrow Exe\ (R_3, R_4, R_5, R_6) = Exe\ (O_1, O_2) \Rightarrow Exe\ (R_3, R_4, R_5, R_6)$

As proved by Theorem 1, $Exe\ (P_1) = Exe\ (O_1, O_2) \Rightarrow Exe\ (R_3, R_4, R_5, R_6)$

So, the results do not change after re-optimization and merge.

Now we send FileScan(Supplies) and FIL(pno=2) for execution and the re-optimization decision is "Yes".

We update the MergeTable

Hash Code of Executed Operator Type	Executed Operator Result
B422ED….// TS(Suppliers)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA 3, 32, Oracle, 13, 5th street, SF, CA …
F2AC13……// TS(Suppliers),FIL(sctiy=Seattle,sstate=WA)	1,32,Boeing,13,1st street,Seattle,WA 2, 31,Amazon, 14, 4th ave, Seattle,WA …
D42423……//TS(Supplies)	1,sugar,2 2, cotton,4 3, rice,7 …

After re-optimization, we have a different plan here:

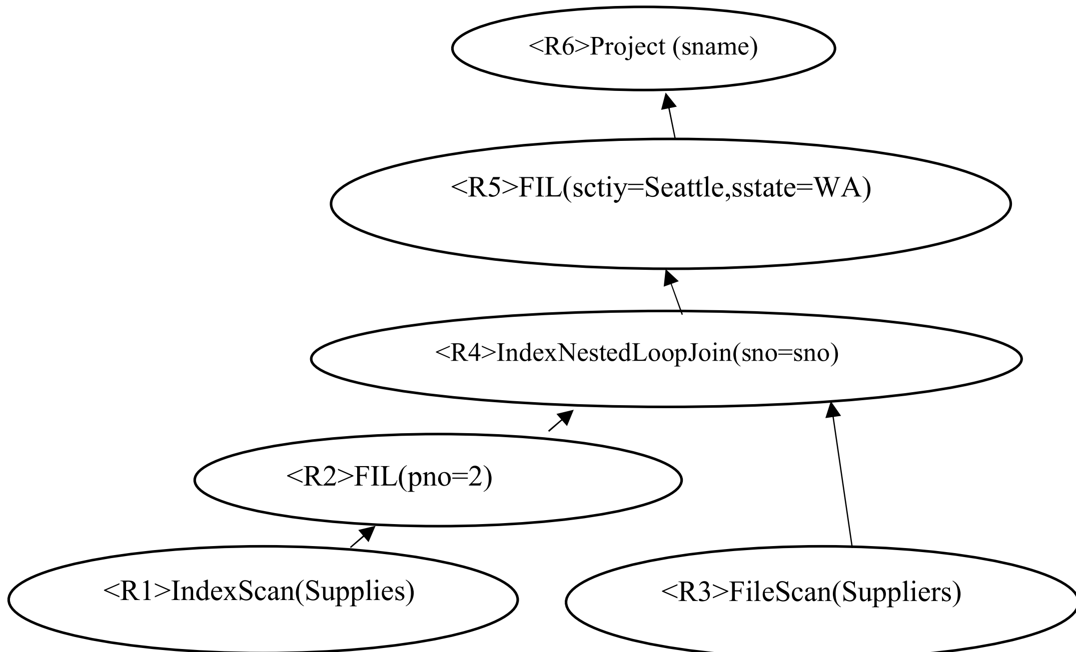


Figure 20. QEP P_2 after 2nd re-optimization

After Merge, we have

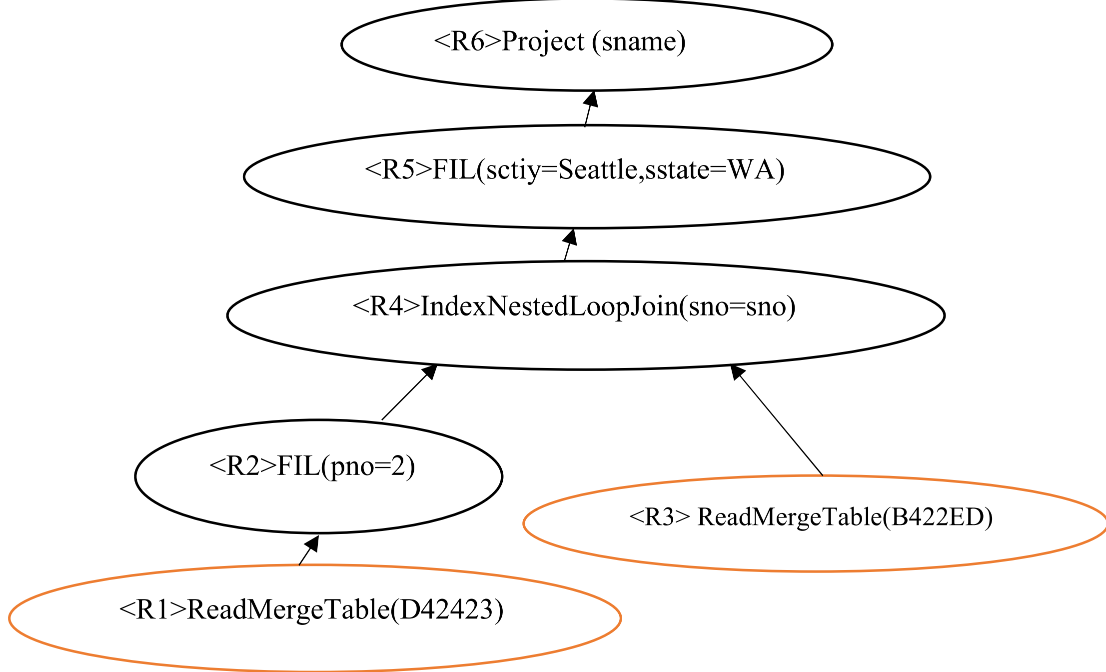


Figure 21. QEP P_2 after operators O_1 and O_3 are merged

Since we cannot find any more hash code matches in the MergeTable, the P_2 is the above. At this point, $P_1 = (O_1, O_2, O_3, O_4, O_5, O_6)$, $P_2 = (R_1, R_2, R_3, R_4, R_5, R_6)$, $P_{k1} = (O_1)$, $P_{k2} = (O_3)$, $P_{q1} = (R_1)$, $P_{q2} = (R_3)$, $Exe\ (P_{k1}) = Exe\ (P_{q2})$, $Exe\ (P_{k2}) = Exe\ (P_{q1})$,

$$Exe\ (P_2) = Exe\ (R_1, R_2, R_3, R_4, R_5, R_6)$$

$$= Exe\ (R_1) \Rightarrow Exe\ (R_2) \Rightarrow Exe\ (R_3) \Rightarrow Exe\ (R_4, R_5, R_6)$$

$$= Exe(O_3) \Rightarrow Exe\ (R_2) \Rightarrow Exe\ (O_1) \Rightarrow Exe\ (R_4, R_5, R_6)$$

According to Theorem 1, if $Exe\ (O_1) = Exe\ (R_3)$

$$Exe(P_1) = Exe\ (R_1, R_2) \Rightarrow Exe\ (O_1) \Rightarrow Exe\ (R_4, R_5, R_6)$$

$$= Exe\ (R_1) \Rightarrow Exe\ (R_2) \Rightarrow Exe\ (R_3) \Rightarrow Exe\ (R_4, R_5, R_6)$$

= Exe(O_3) ⇒ Exe(R_2) ⇒ Exe (O_1) ⇒ Exe (R_4, R_5, R_6)

So, Exe(P_1) =Exe(P_2)

This shows that after re-optimization and merging, the query results do not change, which shows the correctness of the ReOptML algorithm. This process keeps repeating until O_6 is executed and the results are returned to the user.

6.1.1.2 Proof of Correctness of ReOptRL

In this section, we prove that ReOptRL is correct. This means we prove that for the same query, the query result generated by ReOptRL is the same as the query result generated by NoReOpt. We assume that a DBMS that will use our proposed algorithm ReOptRL for query re-optimization will have a correct query optimizer, meaning that this query optimizer will be able to convert an SQL query to a correct logical plan. Then to prove the correctness of ReOptRL, we prove the following theorem 2:

Theorem 2:

Given Plan as the query logical plan produced by an existing query optimizer for a query q, the query result of executing Plan by the two algorithms, ReOptRL and NoReOpt, is the same.

Definition:

Let $L_i$be the logical operator in Plan and let

$$Phy(L_i, K) = P_{ij}^{k}$$

denote the physical operator P_i of type j converted from the logical operator L_i using Algorithm K.

Let

$$Res_i = Exe(P_{ij})$$

denote the result of executing physical operator P_i of type j

Proof:

By using the query optimizer Q to convert a logical operator to a physical operator,

$Phy(L_i, K) \in P_i = \{P_{i1}, P_{i2}, P_{i3}, \ldots, P_{in}\} \forall$ K

Based on [43], $Exe(P_{i1})$=$Exe(P_{i2})$=$Exe(P_{i3})$ … =$Exe(P_{i1})$ if P_{i1}, P_{i2}, P_{i3}, …, $P_{in} \in P_i$,

that is executing the same physical operator with different operator types, the result is the same. Different operator types mean the execution of the operator is implemented using a different algorithm. For example, the join operator can be implemented using NestedLoopJoin or IndexJoin. Both implementations produce the same result.

As ReOptRL and NoReOpt will not generate new physical operator types,

let A be the algorithm ReOptRL and B be the algorithm NoReOpt.

Thus, we have

$Phy(L_i, A) \in P_i = \{P_{i1}, P_{i2}, \ldots, P_{in}\}$ and $Phy(L_i, B) \in P_i = \{P_{i1}, P_{i2}, P_{i3}, \ldots, P_{in}\}$

So,

Res_i= $Exe(P_{im})$= $Exe(P_{in})$

This means the query result of the logical operator L_i is the same whether it is executed using ReOptRL or NoReOpt.

6.1.2 Time Complexity Analysis of ReOpt, ReOptML and ReOptRL

In this section, the time complexity of ReOpt, ReOptML and ReOptRL will be analyzed. We will go through each algorithm line by line and compute the time cost of each step of the algorithm.

6.1.2.1 Time Complexity Analysis for ReOpt

Figure 2 in Section 3.3 shows the algorithm of ReOpt. The variables of the time complexity analysis of ReOpt are listed as follows:

C: constant

Op: total number of operators

N_{attr}: total number of attributes in all the tables

L_{opi}: string length of operator i, e.g., L_{op} of Read= 4

Min_value: a variable used in the Optimization function

Iter_limit: a variable used in the Optimization function

We analyze the time complexity by evaluating the time cost of each line in the algorithm and adding them together as the result.

Line1: Ops ← compile query Sql to get its set of compiler-generated operators

The time cost depends on the cost of a specific query compiler, so we assume the cost is X here.

Line2: Optimizer-Tree ← generate a multi-staged optimizer tree from the set of operators Ops

The length of each operator is L_{opi} and there are Op operators in total, so the cost is Op* L_{opi}

Line 4: G ← map the stage in Optimizer-Tree to form a dataflow graph

The cost of this line is as same as Line2, so the cost is Op*L_{opi}

Line 5: Initial-Schedule ← call function DISPATCH (G, C, CONS) to assign operators to containers to form the initial schedule.

This line assigns each operator in the DAG to one of the containers.

So, the total cost is Op*L_{opi}*N_{cont}

Line 6：Optimized-Schedule ← call function

OPTIMIZE (Initial-schedule, CONS, Min_value, Iter_limit, P) to find the optimized schedule for the initial schedule.

The Min_value controls the number of outer loops in the Optimize function and the Iter_limit controls the number of inner loops in the Optimize function. The worst case is that the Optimize function conducts Min_value* Iter_limit times. For each iteration, as shown in Figure 15 in Section 5.4, in Line 5, the cost of generating a new schedule is L_{opi}. In Line 6, Get_Cost () function iterates the whole schedule. The time cost of this function

is Op* L_{opi}. And then, in Line 10, the time cost of calculating ap is 9*C. Finally, the costs of Line 10 and Line 11 are 2*C.

Finally, the total cost of Line 6 is (11*C+ Op* L_{opi})*Min_value*Iter_limit

Line 7: Result ← execute the current stage of the Optimized-Schedule

The execution cost also depends on the query execution engine. Thus, we assign the cost of this line to Y.

Line 8: Optimizer-Tree ← Eliminate the finished operators from the Optimizer-Tree

The cost of this line is L_{opi}

Line 9: Update constraints and data statistics.

The constraints have 2 variables and data statistics depend on how many attributes are there in the table.

The cost of this line is 2*C+N_{attr}.

The following table gives the summary of the time analyses of all the lines:

Table 4. Line by line time cost of ReOpt

Line No.	**One Time Cost**	**Total Cost After Loop**
Line 1	X	X
Line 2	Op*L_{opi}	Op*L_{opi}
Line 4	Op*L_{opi}	Op^2*L_{opi}
Line 5	Op*L_{opi}*N_{cont}	Op^2*L_{opi}*N_{cont}
Line 6	(11*C+Op*L_{opi})*Min_value* Iter_limit	(11*C+Op*L_{opi})*Min_value* Iter_limit*Op

Line 7	Y	Y*Op
Line 8	L_{opi}	L_{opi}*Op
Line 9	2*C+N_{attr}	(2*C+N_{attr}) *Op

The total cost of the entire algorithm is

$$\begin{aligned}\text{Total Cost} &= X + Op * L_{opi} + Op^2 * L_{opi} + Op^2 * L_{opi} * N_{cont} + (11*C + Op* L_{opi}) * \\ &\quad Min_value * Iter_limit * Op + Y * Op + L_{opi} * Op + (2 * C + N_{attr} * C) * Op \\ &= Op^2 * (1 + N_{cont} * Min_value * Iter_limit) * L_{opi} + Op * (2 * L_{opi} + 11 * \\ &\quad Min_value * Iter_limit + N_{attr} + (2 + Y) * C) * Op + X\end{aligned}$$

The worst-case time complexity of ReOpt is O (Op^2)

6.1.2.2 Time Complexity Analysis for ReOptML

Similarly, we analyze the time cost of ReOptML by going through the algorithm in Figure 10 in Section 4.3.

The variables of the time complexity analysis of ReOptML are listed as follows:

C: constant

Op: total number of operators

N_{attr}: total number of attributes in all the tables

L_{opi}: string length of operator i, e.g., L_{op} of Read= 4

L_{attri}: string length of attribute i, e.g., L_{attri} of “InstructorID” is 12

Line 1: OldStatistics = get current data statistics

The number of data statistics is determined by the total number of attributes in the table. The cost of this line is L_{attri} *N_{attr}.

Line 2: Result = Ø

The cost of this line is C.

Line 3: MergeTable = Ø

The cost of this line is C.

Line 4: QEP = GenerateQEP (OldStatistics, Result, Query, MergeTable)

The time cost depends on the cost of a specific query compiler, so we assume the cost is X here.

Line 5: Result = execute the next available operator or stage if a stage is available in QEP

The execution cost depends on the query execution engine. Thus, we assign the cost of this line as Y.

Line 6: MergeTable = UpdateMergeTable (Result)

The worst case to update the MergeTable is that all the executed operators are recorded as a hash code in Column 1 in the MergeTable. Suppose the time cost of the hash function is Z, so the cost is L_{opi}*Z*Op

Line 7: NewStatistics=UpdateDataStatistics ()

The same as Line 1, the cost of this line is L_{attri}*N_{attr}.

Line 8: DiffStatistics = compute the difference between OldStatistics and NewStatistics

The cost of this line is $2*L_{attri}*N_{attr}$.

Line 10: ReOptDecision = RunPredictiveModel (DiffStatistics)

It depends on the selected model to run. We assume the cost of this line is P.

In the result of the algorithm, the cost of each line is included in the above. So, we give the summary of the time complexity analysis of the whole algorithm in the following table.

Table 5. Line by line time cost of ReOptML

Line No.	**One Time Cost**	**Total Cost After Loop**
Line 1	$L_{attri} *N_{attr}$	$L_{attri} *N_{attr}$
Line 2	C	C
Line 3	C	C
Line 4	X	X
Line 5	Y	Y
Line 6	$L_{opi} *Z*Op$	$L_{opi} *Z*Op$
Line 7	$L_{attri} *N_{attr}$	$L_{attri} *N_{attr}$
Line 8	$2*L_{attri} *N_{attr}$	$2*L_{attri} *N_{attr}$
Line 10	P	P*Op
Line 11	X	X*Op
Line 12	Y	Y*Op
Line 13	$L_{opi} *Z*Op$	$L_{opi} *Z*Op^2$
Line 20	$L_{attri} *N_{attr}$	$L_{attri} *N_{attr}*Op$
Line 21	$L_{attri} *N_{attr}$	$L_{attri} *N_{att}*Op$
Line 22	$2*L_{attri} *N_{attr}$	$2*L_{attri} *N_{attr}*Op$

The total cost of the entire algorithm is

$$\text{Total Cost} = L_{attri} * N_{attr} + C + C + X + Y + L_{opi} * Z * Op + L_{attri} * N_{attr} + 2 * L_{attri} * N_{attr} + P * Op + X * Op + Y * Op + L_{opi} * Z * Op^2 + L_{attri} * N_{attr} * Op + L_{attri} * N_{attr} * Op + 2 * L_{attri} * N_{attr} * Op$$

$$= Op^2 * L_{opi} * Z + Op * (L_{opi} * Z + P + X + Y + 4 * L_{attri} * N_{attr}) + 4 * L_{attri} * N_{attr} + 2 * C + X + Y$$

The worst-case time complexity of ReOptML is O (Op^2)

6.1.2.3 Time Complexity Analysis for ReOptRL and SLAReOptRL

The variables of the time complexity analysis of ReOptRL are listed as follows:

C: constant

Op: total number of operators

N_{attr}: total number of attributes in all the tables

L_{opi}: string length of operator i, e.g., L_{op} of Read= 4

L_{attri}: string length of attribute i, e.g., L_{attri} of "InstructorID" is 12

A_i: total number of attributes used by operator i

N_{cont}: total number of containers

N_{type}: total number of physical operator types supported in a database system

N_{layer}: total number of hidden layers in the neural network (excludes input and output layer)

We analyze the time complexity by evaluating the time cost of each line in the algorithm and adding them together as the final result.

Line 1 : t = 0

This line contains one assignment. The time cost of the assignment is C

Cost of Line 1: C

Line2 : Result = Ø

The cost is the same as that of Line 1.

Cost of Line 2: C

Line 3 : Q_t = 0

The cost is the same as that of Line 1.

Cost of Line 3: C

Line 4 : QEP = QueryOptimizer (query)

Assuming the cost of running the query optimizer in a database system is X.

Cost of Line 4: X

Line 5 : while QEP ≠ Ø

This line contains one comparison, the cost of comparison is C.

Cost of Line 5: C

Line 6 : L_{op}=next available logical operator in QEP

The QEP is stored as a queue in implementation. The next available operator is always stored at the head.

Cost of Line 6: C

Line 7 : State S_t= convert QEP to a state matrix

This step is converting QEP to a state matrix. It can be decomposed to several sub-steps.

7.1 Read the 1st node of the logical plan tree (Node)

Cost of 7.1 is C

7.2 Create an empty entry in the State_Matrix

This entry is an array. As this array needs to hold the operator's name and all the attributes in the tables, the length of this array is N_{attr+1}. Assigning each slot of the array an initialized value costs C. Thus, the total cost is (N_{attr}+1)

Cost of 7.2 is (N_{attr}+1)

7.3 Insert OperatorName at the 1st slot of the array

The cost of this step is equal to the number of characters of this OperatorName

Cost of 7.3 is L_{opi}

7.4 Get the 1st attribute in Node, Node.attributeslist[1]

This step reads the 1st attribute, the cost depends on the length of this attribute.

Cost of 7.4 is L_{attri}*C

7.5 Find the position of Node.attributeslist[1]

This step is to find the position of the attributes in 7.4. It needs to iterate every item in the attribute list. If it matches, then write '1' at that position, otherwise write '0'. Thus, the cost depends on the length of the attributes list.

Cost of 7.5 is N_{attr}*C

For each attribute involved in the operator, Steps 7.4 to 7.5 are repeated. Thus, the total cost of 7.4 and 7.5 is $A_i*(L_{attri}+N_{attr})$

And for each operator in the QEP, Steps 7.1 to 7.5 repeat.

Cost of Line 7:

$Op*(1+ N_{attr}+1+L_{opi}+A_i*(L_{attri}+N_{attr}))$

Note that, for each time, one operator is removed from the QEP. The total number of operators is reduced after the outer loop is executed.

The total cost of the 1st time running of Line 7: $Op*(1+N_{attr}+1+L_{opi}+ A_i*(L_{attri}+ N_{attr}))$

The total cost of the 2nd time running of Line 7: $(Op-1)*(1+N_{attr} +1+ L_{opi}+ A_i*(L_{attri}+ N_{attr}))$

The total cost of the opth time running of Line 7: $1*(1+ N_{attr} +1+ L_{opi}+ A_i*(L_{attri}+ N_{attr}))$

Thus, the cost of running from 1st time to opth time is an arithmetic sequence, and the total cost can be calculated as,

$(Op+1)*Op/2*(1+ N_{attr} +1+ L_{opi}+ A_i*(L_{attri}+ N_{attr}))$

Line 8: Actiont= RunLearningModel (S_t)

This step describes how an action is selected by running the RL model. This step contains several sub-steps:

8.1 Convert all the operators in the current QEP into a numeric value.

The number of operators is Op and converting each operator to a numeric value depends on the length of the operator.

Cost of 8.1: $L_{opi}*Op$

8.2 Read the State matrix as the input of the neural network.

Each node in the input layer of the neural network is corresponding to a value in the State Matrix. Assume we have State Matrix S,

InputNodep=S[i][j].

Each assignment includes a read and an assignment, which costs 2*C

Cost of 8.2: $(1+N_{attr})*Op*2*C$

Similar to Line 7, as the number of operator changes, the total cost of running the whole loop is

$2*C*(1+N_{attr})*Op+(1+N_{attr})*(Op-1)+(1+N_{attr})*(Op-2)+ \ldots +(1+N_{attr})*1$

$= (Op+1)*Op*C*(1+N_{attr})$

8.3 Initialize the weights (W_i).

As a fully connected neural network, the number of weights is equal to the number of nodes in the neural network, and the number of nodes in the hidden layer is the same as the number of nodes in the input layer. The nodes of the output layer are the same as the number of actions. The number of actions is the number of containers * the number of operator types. The total number of nodes is calculated as the number of nodes of the input layer + the number of nodes of the hidden layers + the number of nodes of the output layer. The cost of assigning a value to weight is C,

Cost of 8.3: $C*(1+N_{attr})*Op*(N_{layer}+1)+N_{cont}*N_{type}$

8.4 Calculate the node value of the hidden layer 1:

8.4.1 Calculate the value of the 1st node in the hidden layer 1

$InputNode_1*W_1+InputNode_2*W_2+\ldots+InputNode_{op}*(1+N_{attr})*W_{op}*(1+N_{attr})$

Cost of 8.4.1: $2*(Op*(1+N_{attr}))$

8.4.2 Calculate the value of the 2nd node in the hidden layer 1

These steps repeat till all the nodes in the hidden layer 1 is finished

Repeat $Op*(1+N_{attr})$ times

Cost of 8.4: $2*C*(Op*(1+N_{attr}))^2$

8.5 Calculate the node value of the hidden layer 2

Similarly, total cost of 8.5: $2*C*(Op*(1+N_{attr}))^2$

These repeat for all the hidden layers.

The cost of all the hidden layers: $2*C*(Op*(1+N_{attr}))^2*N_{layer}$

8.6 Calculate the node value of the output layer:

Similarly, Total Cost of 8.6: $2*C*(Op*(1+N_{attr}))*N_{cont}*N_{type}$

Total Cost of calculating the values of all the nodes: $2*C*(Op*(1+N_{attr}))^2*N_{layer}$

$+2*C*(Op*(1+N_{attr}))*N_{cont}*N_{type}$

8.7 Find the action with the max q-value

In the output node layer, every node contains a key-value pair <Action, Q-value>

e.g., <(Read, 4), 4.45>, <(Read,5), 3.1>,

The following sub-step is to find the action with the max Q-value:

8.7.1 Set max=0

Cost of 8.7.1: C

8.7.2 Read the key-value pair and the Q-value as the current Q-value

Cost of 8.7.2: $C*(L_{opi}+1)$

8.7.3 if the current Q-value > max, then max=q-value

Cost of 8.7.3: C+C

Steps 8.7.2 to 8.7.3 repeat for $N_{cont}* N_{type}$ times

Cost of 8.7: $N_{cont}* N_{type} *(2C+ C*(L_{opi}+1))+C= (N_{cont}* N_{type}+1)*(3+ L_{opi})$

From Line 8.1 to Line 8.7, we can find the Cost of Line 8:

$L_{opi}+(1+ N_{attr})*Op +2*(Op*(1+ N_{attr}))^2* N_{layer}$

$+2*(Op*(1+ N_{attr})) * N_{cont}* N_{type}+ (N_{cont}* N_{type}+1)*(3+ L_{opi})$

Line 9: Result=Result ∪ execute (Op, $Action_t$)

Assuming the cost of executing one query operator is Y.

Cost of Line 9: Y

Line 10: QEP= QEP - Op

The QEP is stored as a queue. Remove the Op is a dequeue operation in implementation.

The detail of this dequeue function is as follows,

```
function dequeue () {
  lop = head.value        // c
   head = head.next          // c
   size--                  // 2c
   if (head == null) {      // c
      tail = null           //c
   }
   return lop       // c
}
```

The cost of each line is also denoted at the end, and the accumulative cost is C+C+2C+

+C+C=7C

Cost of Line 10: 7C

Line 11: Update R_t=R (wp, $Action_t$.time, $Action_t$.money))

The reward function is the following:

$$Reward\ R = \frac{1}{1 + (W_t * (T_{op}^q + P_t)) + (W_m * (M_{op}^q + P_m))}$$

As one arithmetic operation cost C,

Cost of Line 11: C+(C+C)+C+(C+C)+C= 6C

Line 12: Obtain the Q-value of the next state Q_{t+1} from the neural network

This step has the same cost as the cost in 8.7.2

Cost of Line 12: C*(L_{opi}+1)

Line 13：Update Q-value of current state Q_t = Bellman (Q_t, Q_{t+1}, Rt, α, ϒ)

Similar to Line 11， as one arithmetic operation costs C,

Cost of Line 13: C+(C+C)+C+(C+C)+C+C+C=9C

Line 14：Update Weights in the neural network

The cost of updating the weights is the same as the cost of initializing the weights in Line 8.3.

Cost of Line 14: $C*(1+N_{attr})*Op *(N_{layer}+1) + N_{cont}* N_{type}$

Line 15：t=t+1

Cost of Line 15: 2C

From Line 6 to Line 15, each line is in the WHILE loop. Thus, to calculate the overall cost of the whole algorithm, the cost of those lines should be computed in total considering the loop.

The following table gives the summary of all the lines,

Table 6. Line by line time cost of ReOptRL

Line No.	One time cost	Total Cost After Loop
Line 1	C	C
Line 2	C	C
Line 3	C	C
Line 4	X	X
Line 5	2C	2C

Line 6	C	C*Op
Line 7	$Op*C*(1+ N_{attr} +1+ L_{opi}+ Ai*(L_{attri}+ N_{attr}))$	$(Op+1)*Op^2/2* C*(1+ N_{attr} +1+ L_{opi}+ A_i*(L_{attri}+ N_{attr}))$
Line 8	$L_{opi}+(1+ N_{attr})*Op +2*(Op*(1+ N_{attr}))^2* N_{layer}+2*(Op*(1+ N_{attr})) * N_{cont}* N_{type}+ (N_{cont}* N_{type}+1)*(3+ L_{opi})$	$(Op+1)*Op/2*(L_{opi}+(1+ N_{attr})*Op +2*(Op*(1+ N_{attr}))^2* N_{layer}+2*(Op*(1+ N_{attr})) * N_{cont}* N_{type}+ (N_{cont}* N_{type}+1)*(3+ L_{opi}))$
Line 9	Y	Y*Op
Line 10	7*C	7*C*Op
Line 11	6*C	6*C*Op
Line 12	$C*(L_{opi}+1)$	$C*(L_{opi}+1)*Op$
Line 13	9*C	9*C*Op
Line 14	$C*(1+ N_{attr})*Op *(N_{layer}+1) + N_{cont}* N_{type}$	$C*(1+ N_{attr})*Op *(N_{layer}+1) + N_{cont}* N_{type}*Op$
Line 15	2C	2C*Op

So, the total cost of the entire algorithm is

$$\text{Total Cost}= 5C + X + (C + Y) * Op + C * (Op*(1 + N_{attr} + 1 + L_{opi} + A_i*(L_{attri} + N_{attr})) + (L_{opi} + (1 + N_{attr}) * Op +2 * (Op * (1 + N_{attr}))^2* N_{layer}+2 * (Op * (1 + N_{attr})) * N_{cont}* N_{type} + (N_{cont}* N_{type} + 1) * (3 + L_{opi}))$$

$$= Op^2 * (2*(C + N_{attr}))^2) + (Y + (4C + A_i + L_{opi}) * N_{attr} + 7C + L_{opi}) * N_{cont} * N_{type} + L_{opi} * (A_i * L_{attri} + 1)) * Op + 5C + X$$

The worst-case time complexity of ReOptRL is O ($N_{attr}^2 * Op^2$). The only difference between SLAReOptRL and ReOptRL in terms of time complexity is that, in Line 11, the cost is 13*C. The worst-case time complexity of SLAReOptRL is also O ($N_{attr}^2 * Op^2$).

6.2 Experimental Results

6.2.1 Experimental Hardware and Software Configurations and Benchmark Dataset

There are two sets of machines (containers) that are used in our experiments. The first set consists of a single local machine used to train the machine learning model and to perform the query optimization. This local machine has an Intel i5 2500K Dual-Core processor running at 3 GHz with 16GB DRAM. The second set consists of 10 dedicated Virtual Private Servers (VPSs) that are used for the deployment of the query execution engine. 5 of these VPSs are called small containers, each of which has an Intel Xeon E5-2682 processor running at 2.5GHz with 1 GB of DRAM. The other 5 VPSs are called large containers, each of which has two Intel Xeon E5-2682 processors running at 2.5GHz with 2 GB of DRAM. The query optimizer and the query engine used in this experiment are modified from the open-source database management system, PostgreSQL 8.4 [44]. The data are distributed among these VPSs.

The queries and database tables are generated using the TPC-H database benchmark [36]. There are eight database tables with a total size of 1,000 GB and the database tables are

populated using the default data generator. We run 50,000 queries in total and these queries are generated by the query templates randomly selected from the 22 query templates from the benchmark. In the experiments, we set the query operator impact rate α_{join} to 1.5 for the JOIN operator and α_{op} to 1 for other operators.

6.2.2 Competitive Algorithms

To evaluate the performance of the four proposed algorithms, ReOpt, ReOptML, ReOptRL, and SLAReOptRL, the following existing algorithms are selected for experimental comparison studies:

1) **NoReOpt:** This is an algorithm that uses no re-optimizations at all. There are multiple query processing algorithms on the market. We use the one that is applied in the original Postgre SQL. NoReOpt is also considered as the "Baseline" when comparing different algorithms.

2) **Tukwila** [15]**:** This algorithm is a well-known adaptive query re-optimization algorithm in the literature that triggers a re-optimization after an operator is executed if the difference between the estimated query cost and the actual query cost exceeds some threshold. The details of this algorithm were discussed in Section 2.1.1.

3) **Sample** [3]**:** This is an algorithm existing in the literature where query re-optimization uses sampling-based query estimation. Unlike traditional query re-optimizers, the estimation of executing each query is done by estimating a sample of the entire dataset so

that the speed of estimating execution cost is faster. The details of this algorithm were described in Section 2.1.3.

We choose Tukwila as one of the competitive algorithms because we would like to study query execution performance when the query re-optimization decision is made by different methods. The workflow of Tukwila is similar to that of ReOptML. They both start with a QEP generated by an existing query optimizer. After executing one or a stage of query operators, the decision of whether or not to conduct query re-optimization needs to be made. In Tukwila, if the difference of data statistics before and after execution is greater than a threshold, the re-optimization is triggered. The threshold determines when a query re-optimization should take place, but it needs to be set by domain experts, while in ReOptML, a supervised learning technique is applied to make such decisions without any human interference. We compare ReOpt, ReOptML, and Tukwila to investigate the difference in query execution performance when query re-optimization is conducted without any decision, with the decision made by the machine alone, and with the decision made by humans, respectively.

Also, we choose Sample as another competitive algorithm because we would like to study the impact of the time overhead from updating the data statistics on the overall query execution performance. In Sample, after executing a sample of tuples, the column cardinalities are updated. The query optimizer uses the new cardinality to re-optimize the remainder of the query execution plan. The other data statistics such as histogram and the

number of rows are not updated in re-optimization. Thus, in this technique, the time overhead from updating the data statistics is reduced. However, potentially, the overall performance may not be improved as a bad QEP still can be generated after re-optimization using only the updated cardinalities. To avoid the time overhead caused by updating data statistics completely, ReOptRL and SLAReOptRL are designed. We would like to see if the query execution performance can still be improved even without using any updated data statistics. Thus, we compare the four proposed algorithms with Sample to investigate the query execution performance when query re-optimization relies on fully updated data statistics (ReOpt and ReOptML), partially updated data statistics (Sample), and no updated data statistics (ReOptRL and SLAReOptRL).

Meanwhile, there exist more recent query re-optimization techniques other than Sample. Those are SkinnerDB [5], CuttleFish [4], and ReJoin [6]. We do not choose them as competitive algorithms in the experiments because there exist restrictions to use each of those techniques. For SkinnerDB, it assumes that the existing query optimizer only generates the left-deep trees for join operators, while in our experiments, besides the left-deep trees, the query optimizer can also generate the right-deep trees and the bushy trees. To compare with SkinnerDB, we have to restrict our query optimizer to generate the left-deep trees only. This largely narrows the search space of QEPs and thus an optimal QEP is more likely not chosen after re-optimization. The query execution performance in our proposed algorithms is negatively impacted if a sub-optimal QEP is executed.

For CuttleFish and ReJoin, those two techniques focus only on re-optimizing the join order and physical join operator. Although re-optimizing the join operator is very critical to the query execution performance, other factors such as resource provisioning and execution order of query operators are also important but are not re-optimized in these algorithms. Thus, those techniques perform well only for executing queries that contain a high number of join operators, while the queries in our experiments contain both a low and high number of join operators. To compare with CuttleFish and ReJoin, we must restrict to executing queries that contain a high number of joins only. Thus, the query execution performance results are biased toward those two algorithms.

If we implement those restrictions in our proposed algorithms, those unchosen competitive algorithms may outperform our proposed algorithms on query response time or monetary cost. This is because the unchosen competitive algorithms are designed specifically for re-optimizing join operators while our algorithms are designed for re-optimizing the entire query execution plan which can consist of other operators, such as read, filter, and aggregation, besides join. However, since none of the unchosen competitive algorithms considers SLA violation, we expect our proposed algorithms to have lower SLA violation rates than those algorithms do.

6.2.3 Performance Metrics

In this section, the performance metrics used in our experimental evaluations of ReOpt, ReOptML, ReOptRL and SLAReOptRL are presented.

6.2.3.1 Performance Metrics for ReOpt

The performance of ReOpt is measured based on two metrics: (1) average response time of a query and (2) average monetary cost to pay to the cloud service provider to execute a query. Query response time is the elapsed time from the moment when a user enters an SQL query to a cloud DBMS until the moment when the results of executing this query are displayed on the screen. The average query response time is the main performance metric when evaluating a query processing algorithm. The queries used in the experiments are generated from the TPC-H benchmarks [36]. There are 22 query types in total. Among those query types, some of them are simple query types and some of them are complicated. Multiple queries generated by the same type are evaluated. Hence, the average response time per query to give a general overview of the performance of each algorithm. Similarly, we use the average monetary cost per query to evaluate how much money should a user pay to complete a query. As in the scenario of this dissertation, all the queries are executed on a cloud database system, we need to consider monetary cost in addition to query response time.

6.2.3.2 Performance Metrics for ReOptML

The performance of ReOptML is also measured based on average query response time and average monetary cost.In addition, in order to select the best supervised machine learning model to be used to predict when a re-optimization should be triggered, model accuracy is also used as the metric for selecting the model. The model accuracy is measured by the number of correct re-optimizations / the total number of re-optimizations.

6.2.3.3 Performance Metrics for ReOptRL and SLAReOptRL

Similarly, the performance of ReOptRL is also measured based on average query response time and average monetary costs. In addition, in SLAReOptRL, which is an extended algorithm of ReOptRL, we measure the SLA violation rate. The SLA violation rate is the total number of queries executed that violate the SLA requirements divided by the total number of queries executed. Using this metric, we can see whether the rate of SLA violation is improved if the algorithm considers SLA requirements while a query is processed.

6.2.4 Experimental Results for ReOpt

In this section, the evaluation results for ReOpt are presented. There are two sets of results, and each set of results is based on running 1140 queries generated from two different query types obtained from the work in [1]. The first set of results aims to study the impact of data size on different degrees of parallelism and the second set of results aims to study the impact of data size on different physical operators.

6.2.4.1 Comparison of Query Response Time and Monetary Cost of ReOpt and NoReOpt on Different Degrees of Parallelism

We hypothesize that query re-optimization is able to reduce the degree of parallelism of the query execution plan which means the query response time and monetary cost will be reduced as fewer computational nodes are planned to be used. The example Query 1 given below is executed to test whether the time cost or monetary cost will be affected by the

degree of parallelism of the query execution plan as the number of containers will be used in the query execution will be impacted. In this query, there are sub-queries that select data from different partitions of the table and are executed in parallel, so there is a high degree of parallelism in this query.

Query 1:

```
SELECT  pid,RecursiveUDA(hr) AS sb
FROM (SELECT pid, hr, FROM patient_1
          UNION
          SELECT pid, hr, FROM patient_2
          UNION
          SELECT pid, hr, FROM patient_3
          ) AS R
WHERE UDF(pid,hr)>80
GROUP BY pid
```

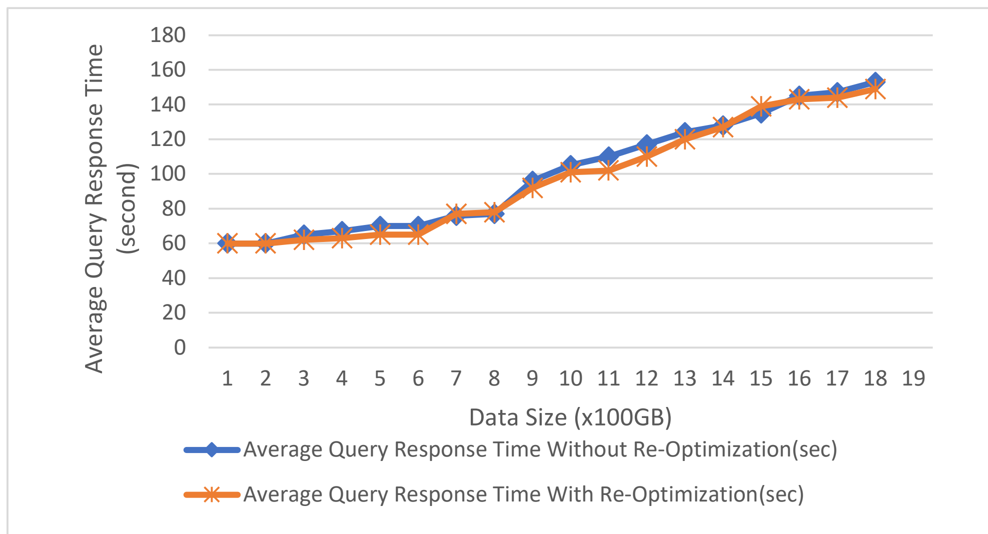


Figure 22. Impact of data size on query response time of Query 1

We expect to see if re-optimization is applied, such change will be detected, and applying this new data size in the rest stages of the query optimization and execution can result in the change in the number of containers used accordingly. (Shown in Table 7). And evidently, the monetary cost will also be changed.

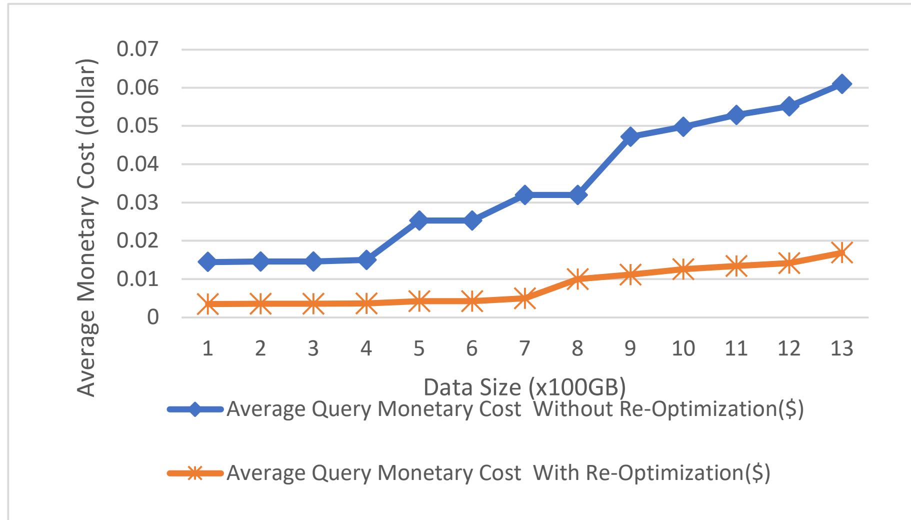


Figure 23. Impact of data size on query monetary cost of Query 1

From Figure 23, we can see that the monetary cost of the query execution has been reduced with the re-optimization while the query response time does not change much, only within 8%, while the query monetary cost is reduced over 40% averagely. This is because the degree of parallelism is updated after each stage; if the degree of parallelism is small, we will schedule fewer containers for the rest of the QEP execution. Thus, the monetary cost is reduced. For example, when the data size is over 1,000 GB, the peak number of containers needed without re-optimization is 8 but this number is reduced to 2 with re-optimization. As 6 fewer containers are used in query execution, this saves the monetary cost.

Table 7. The comparison of the peak number of containers used in execution of Query 1

Data Size (x100GB)	The peak number of Containers used without Re-Optimization	The peak number of Containers used with Re-Optimization
1	4	2
5	4	2
10	8	2
15	8	2
20	8	2

6.2.4.2 Comparison of Query Response Time and Monetary Cost of ReOpt and NoReOpt on Different Physical Operators

In this experiment, we study the impact of physical operators and the impact of different data sizes on these physical operators on our proposed algorithms. The physical operators will be changed with different data statistics even with the same logical operators in the QEP. Similarly, if the QEP is not re-optimized, this change will not be detected before the query is executed and the statistics are updated. To reflect this change, we purposely change the type of the Join operator during the query execution. To test this impact, we run the following Query 2:

Query 2:
```
SELECT  R.p_id,R.p_name,R.sc,S.p_hr
FROM (SELECT p_id, p_name, AVG(p_bp) AS sc
          FROM patient  GROUP BY p_id,p_name) AS R
JOIN (SELECT p_id,p_hr
```

```
            FROM patient
            WHERE UDF(p_id,p_hr)>80
          ) AS S
ON R.p_id=S.p_id
```

In this query, there is a Join of two subqueries and the data size of each subquery is unknown. We want to see how the physical operator of this Join will change depending on the data size of the subquery. So, we purposely make the data size of the right side of the join operator to be small enough to fit in the cache so that the Shuffle Join operator will be changed to the Broadcast Join operator for every query execution. As seen from Figure 24, when the physical operator of Join is changed from Shuffle Join to Broadcast Join, the execution time is reduced as the BroadCast Join is executed around 40% faster than Shuffle Join in this experiment environment. The overall time cost using re-optimization has an average of around 20% improvement over without using re-optimization. Also, as shown in Figure 24, the bigger the data size, the more time is saved with re-optimization as opposed to without query re-optimization even though both approaches will require more time for query execution. This shows that re-optimization is worth for large data size. The monetary cost between the two approaches is close, with only a 4% difference as shown in Figure 25. This difference in monetary cost happens when some part of the query is executed on the containers with a higher unit price after re-optimization.

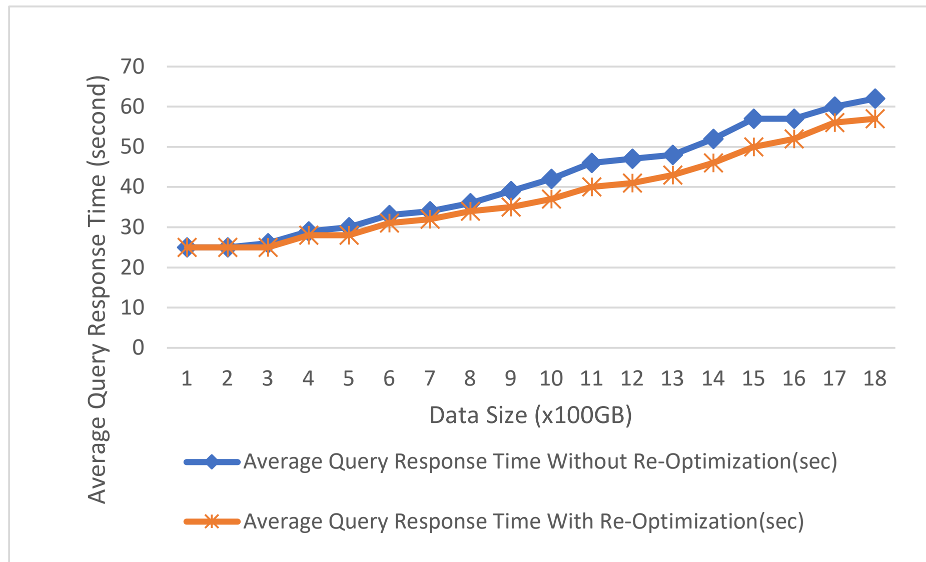


Figure 24. Impacts of data size on time for executing query

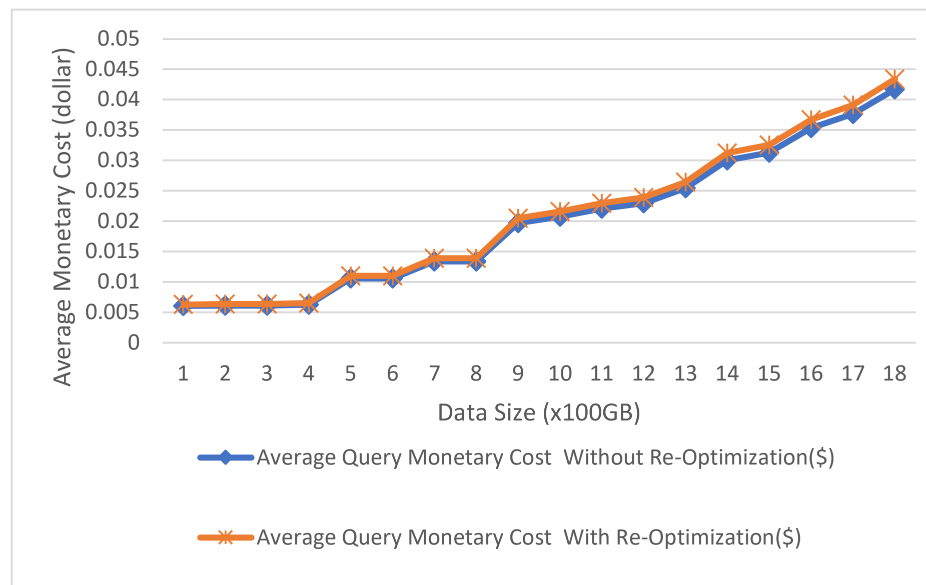


Figure 25. Impacts of data size on monetary cost for executing query

6.2.5 Experimental Results of ReOptML

In this section, the evaluation results are presented using ReOptML. First, we show the performance of each supervised machine learning model. Then we compare the average

query response time and monetary cost among multiple algorithms running on both uniform and skew distributed data after applying the selected model.

6.2.5.1 Comparison of Accuracy of Different Supervised Learning Models

Model accuracy reflects the overall success rate of predicting useful re-optimizations. We use 10-fold cross-validation to test the accuracy of three supervised learning models, Neural Network, Random Forest, and SVM. We also study the impact of different data distributions on the accuracy of the learning models. We populate the database tables with both the uniformly distributed data and skew data and the same queries are executed on both of them. Many traditional query optimizers, like PostgreSQL [44], assume that data is uniformly distributed, so if only uniformly distributed data is used, there are more chances that re-optimization has no effect at all. Skew data may cause wrong cost estimations and thus the QEP selected by the traditional query optimizer is far from optimal, thus re-optimization may be more useful when data is skewed. We use skew data on purpose to see how model accuracy and query execution performance are impacted. As shown in Figure 26, as the number of queries increases, the accuracy increases as well. This is because as more observations were learned by the model, it is more capable of predicting beneficial re-optimizations. We find the accuracy among these three models is slightly different. Averagely, the Neural Network is near 70% accurate, while Random Forest and SVM are close to 75%. From the data distribution perspective, the models on

the uniform data and on the skew data have slightly different accuracies with the average accuracy being within 5% difference of each other.

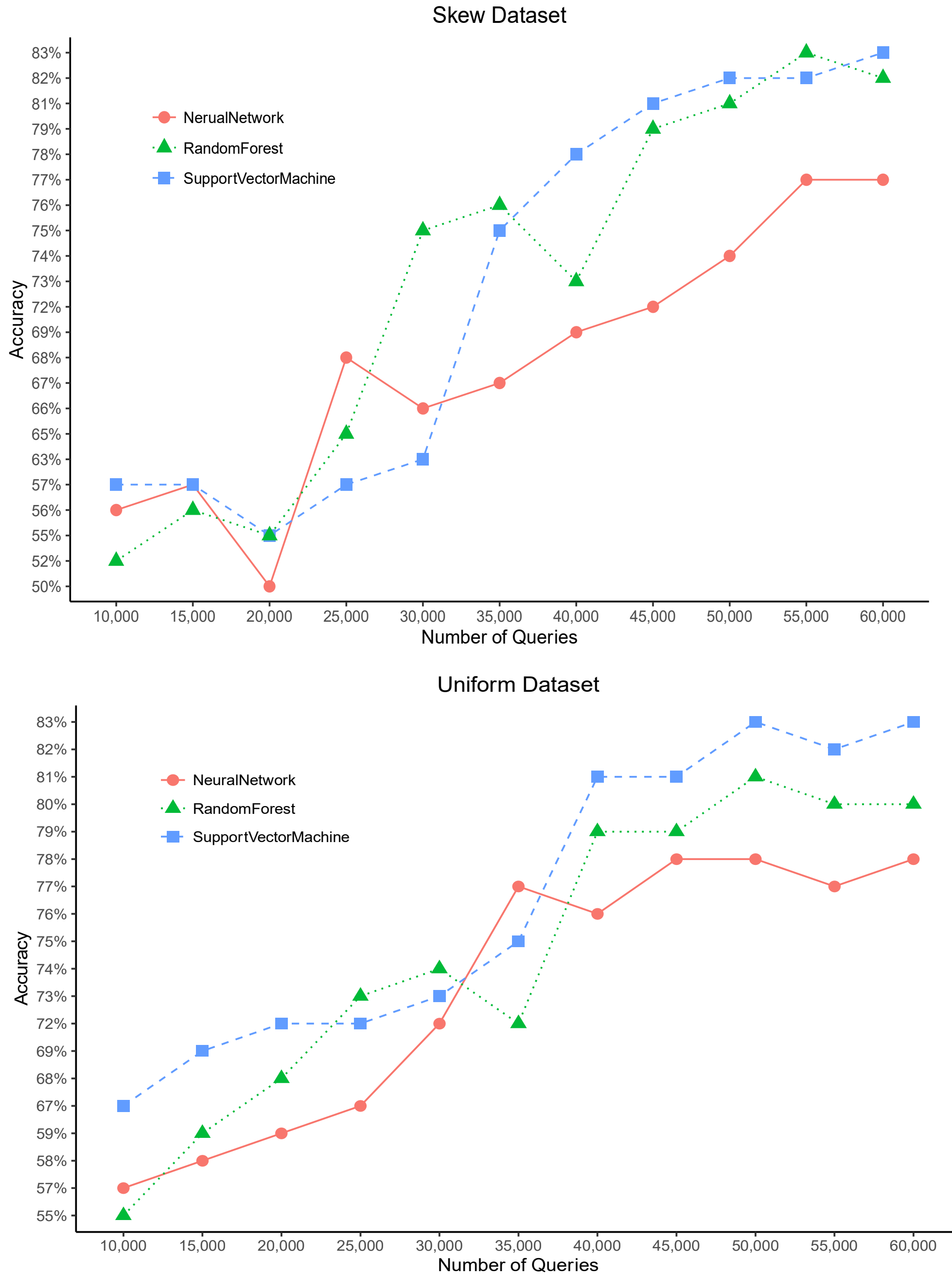


Figure 26. Model accuracy of three different machine learning algorithms that learn from queries executed on (a) uniform data and (b) skewed data

6.2.5.2 Performance Obtained When Applying Different Supervised Machine Learning Models for Query Re-Optimization to Query Processing

The model accuracy is close to each other as reported above; so, to select which supervised learning model should be used eventually, in this section, we evaluate these models in terms of performance on query execution when incorporating them into query processing as shown in Figure 26. We generate 100 query instances from each of the 22 TPC-H benchmark query types, totaling 2200 queries. On average, each query has 13 stages. These queries are executed and re-optimized based on the decisions made by these three models. Each QEP is evaluated with the same weight on time and monetary costs when the query optimizer selects the best QEP. This means we assume the users have no preference on time or monetary costs themselves. The actual time and monetary costs resulting from applying these three models are compared. To be fair, these queries are newly generated and not seen by any of these models during the model training process. Figure 27 shows the end-to-end query response time and monetary cost on executing the queries generated from all 22 query types of the TPC-H benchmark and these costs are summarized in Table 8. These results are averaged on running queries on both uniform and skew datasets.

Table 8. Average and cumulative query response time and monetary cost using three different machine learning models

	Neural Network	Random Forest	SVM
Average Query Response Time	36.2 sec	35.4 sec	31.5 sec
Cumulative Query Response Time of 2,200 queries of 22 query types	79,200 sec	77,880 sec	69,300 sec
Average Query Monetary Cost	0.070 ¢	0.071 ¢	0.069 ¢
Cumulative Query Monetary Cost of 2,200 queries of 22 query types	154.6¢	156.2 ¢	151.8 ¢

From Table 8, we can see that SVM gives the best query response time. As shown in Figure 20, the three models have a very similar model accuracy. This means that the optimizer has a similar chance to perform useful re-optimizations by using any of these models. However, it takes different amounts of time to apply these models. As these models are applied online during query execution, the overheads caused by using these models are added to the query response time. Thus, a small difference in this overhead may cause a significant difference in query response time, and thus is crucial to the users. From the monetary cost perspective, the amount of money to execute each query seems negligible when using any of the three models as shown in Table 8. However, this amount shown in this figure is just for one query execution, but in practice, tens of thousands of queries are executed for enterprise applications. This results in a large difference in cumulative monetary costs. Also, for each query type, the monetary cost has a larger variation than the query response time. This is because, in our hardware configuration, a large container is charged 4 times more money

than a small container according to our price model. If an operator is assigned to a large container, it costs way more money to be executed but the time cost may be just a little bit less. Thus, the accumulative monetary cost varies a lot. Overall, SVM has the best prediction accuracy and query response time, and the second-best monetary cost. Thus, in the following experiments, we select SVM as the machine learning model to be used in our proposed machine learning-based query re-optimization, ReOptML, and compare this algorithm against other query re-optimization algorithms. We select this model for comparison purposes only; we do not intend to suggest which model should be selected automatically as some QEPs may be executed faster but cost more money and vice versa, depending on the selected model.

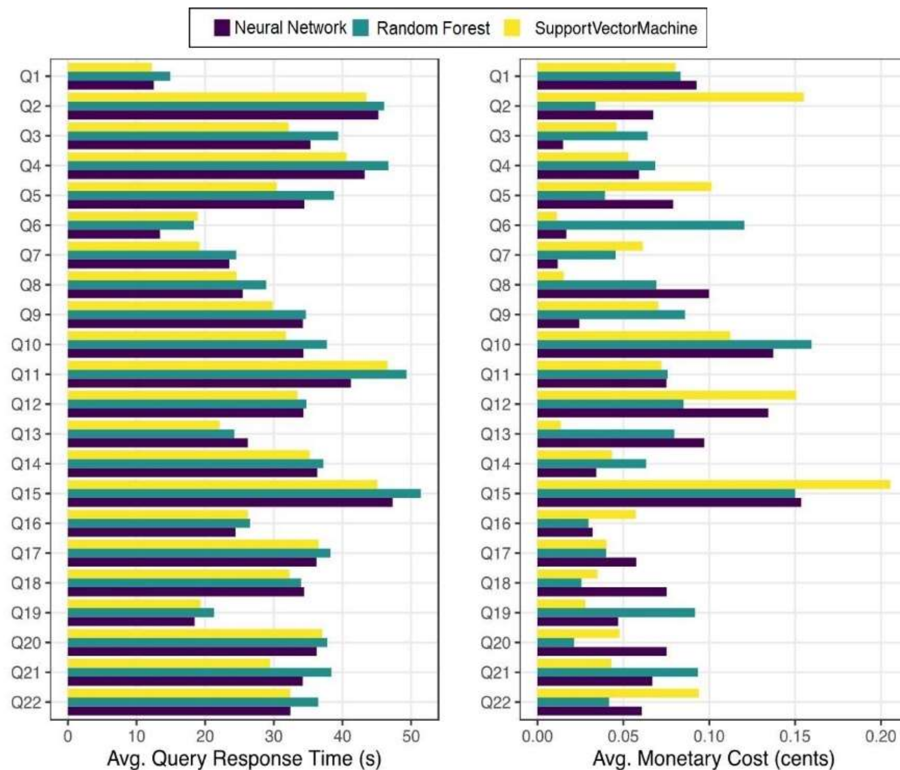


Figure 27. (a) and (b) Average response time and Average monetary cost of executing queries using three different machine learning models for query re-optimization

6.2.5.3 Performance of Different Query Re-Optimization Algorithms

In this section, we compare the end-to-end query processing performances obtained when the following query re-optimization algorithms are incorporated into query processing: 1) the proposed algorithm in this chapter (ReOptML); 2) the algorithm proposed in Chapter III (ReOpt) where a query re-optimization is conducted automatically after the execution of each stage in the query is completed; 3) the algorithm proposed by Tukwila [15] (denoted as Tukwila), a well-known adaptive query re-optimization algorithm that triggers a re-optimization after an operator is executed if the difference between the estimated query cost and the actual query cost exceeds some threshold; and (4) the baseline algorithm where queries are processed without any query re-optimization (denoted as NoReOpt).

We launch 2200 queries with 100 queries being generated from each of the 22 TPC-H query types both on uniform and skew data. We compare the average query response time and monetary cost. We report the query types that have large differences between ReOpt and ReOptML on average so that we can see with the help of machine learning, how much improvement can be obtained with re-optimization.

Skew Data: we compare our algorithm with Tukwila, NoReOpt, and ReOpt on skew data. The experimental results show that our algorithm performs the best both in terms of query response time and monetary costs. From Figure 28 (a), on average we see that ReOptML yields 13%, 22%, and 35% less query response time than ReOpt, Tukwila, and NoReOpt,

respectively. From Figure 28 (b), on average we see that ReOptML spends 17%, 34%, and 35% less monetary cost than NoReOpt, ReOpt, and Tukwila, respectively.

The above results show that ReOptML saves more time and monetary cost than the other three algorithms, ReOpt, Tukwila, and NoReOpt. In this experiment, re-optimization contributes to these savings, and it is beneficial in two aspects. First, after a re-optimization, the optimizer implements different types of physical operators. Different types of physical operators, such as NestedLoopJoin or HashJoin, used to execute these JOINs can result in a large difference in query response time. Second, re-optimization helps decide the degree of parallelism of each operator so that a lot of money is saved as fewer containers are used for executing these operators. However, not all re-optimizations are useful as discussed in Section 4.1, conducting more useful re-optimizations, and avoiding unnecessary re-optimizations can further improve performance. We compare the QEP before and after re-optimization in each algorithm to find out whether each re-optimization is necessary or not. In this experiment, nearly 70% of the re-optimizations are necessary in ReOptML, while only 35% in ReOpt and 28% in Tukwila are necessary. From this, we conclude that using machine learning further helps improve both the time and monetary costs of query execution by avoiding unnecessary re-optimizations.

Uniform Data: In addition to the results obtained from executing queries on skew data, Figures 28 (c) and (d) also show the results of executing the same queries on uniform data. These two figures report only the query types that have large differences in query response

time and monetary cost. From Figure 28 (c), on average we see that ReOptML yields 13%, 13% and 21% less query response time than ReOpt, Tukwila, and NoReOpt, respectively. The total savings of query response times resulting from ReOptML, ReOpt, Tukwila, and NoReOpt on uniform data are less than those on skew data because the optimizer assumes the data is uniformly distributed by default. Thus, the error of cost estimation on uniform data is less than that on skew data. This shows that query re-optimization, in general, is more helpful in executing queries on skew data. In terms of monetary cost, from Figure 28 (d), on average we see that ReOptML spends the same amount of money as ReOpt, 7% less money than Tukwila, but 10% more money than NoReOpt. From these results, we find that when queries are executed on uniform data, re-optimization saves time but does not improve monetary cost.

In summary, we conclude that using supervised machine learning to predict when a re-optimization is beneficial does improve query response time no matter queries are executed on a uniform or skew data. In terms of monetary cost, this algorithm also saves a significant amount of monetary cost when queries are executed on skew data, but gives no improvement when queries are executed on uniform data.

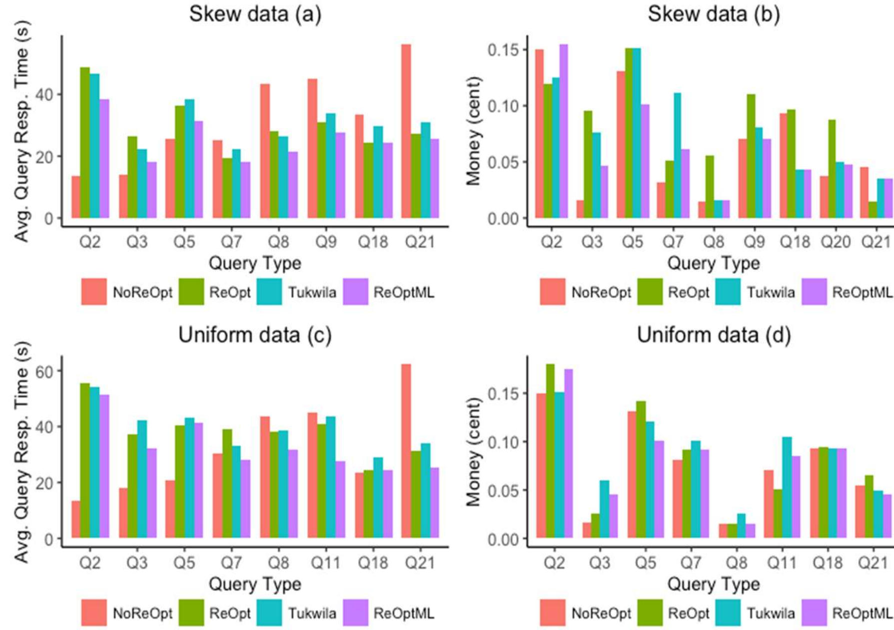


Figure 28. (a)-(d) Average query response time and monetary cost of executing one query from different query types on skew data (a-b) and on uniform data (c-d)

6.2.6 Experimental Results of ReOptRL and SLAReOptRL

In this section, the evaluation results of ReOptRL and SLAReOptRL are presented. We show the evaluation of the two algorithms with and without SLA requirements. Then we show the impact of RatioJoin and Weight on executing queries using different algorithms.

6.2.6.1 Evaluation of ReOptRL and SLAReOptRL with SLA Requirements

In this section, we compare the performance results obtained when the following query re-optimization algorithms are incorporated into query processing: 1) our two proposed algorithms, ReOptRL and SLAReOptRL; 2) the ReOpt algorithm proposed in Chapter III where a query re-optimization is conducted automatically after the execution of each

operator in the query is completed; 3) the ReOptML algorithm proposed in Chapter IV where a query re-optimization is conducted by a supervised machine learning model decision . In this algorithm, after a query operator is executed, conducting re-optimization or not is decided by a supervised machine learning model. This decision is influenced by the current data statistics such as column selectivity and histogram. The re-optimization is done by the traditional query optimizer. Only whether to trigger the re-optimization or not is decided by the supervised learning model; 4) the existing algorithm proposed in [3] where query optimization uses sampling-based query estimation (denoted as Sample), and 5) the existing algorithm that uses no re-optimization (denoted as NoReOpt).

In these experiments, we use NoReOpt as the baseline and the other algorithms are compared to the baseline. Moreover, SLA requirements are implemented. We assign each query with its SLA requirements and the query is executed using different query re-optimization algorithms with the same SLA requirements. Because the SLA requirements depend on the cloud service providers' agreements with their users, there are different ways to define the SLA requirements [37]. In our experiments, we manually set the SLA requirements as the mean value of query response time and the mean value of monetary costs to execute the queries. These mean values are the average query response times and monetary costs obtained when executing 300,000 tested queries, which are the 50,000 queries executed using each of the six studied algorithms. These mean values are the same for all the queries used in the following experiments.

From Figures 29 and 30, we can see that, for both the query execution time and monetary costs, on average SLAReOptRL performs the best and ReOptRL performs the second-best among all the algorithms. Specifically, comparing with the baseline NoReOpt where no re-optimization is conducted, the query execution time improvement using SLAReOptRL is 45%, ReOptRL 39%, ReOptML 27%, ReOpt 13%, and Sample 10%, while the monetary cost improvement using SLAReOptRL is 62%, ReOptRL 52%, ReOptML 27%, ReOpt 17%, and Sample 5%. The above results show that when considering all the 50,000 queries generated from all the 22 TPC-H benchmark query types, compared with the baseline NoReOpt, on average our proposed algorithms improve more time and monetary costs than the three algorithms, ReOpt, ReOptML, and Sample. Especially, the monetary cost has a significant improvement (SLAReOptRL and ReOptRl are 62% and 52% better than NoReOpt, respectively). However, for the queries of simple query types (Q1, Q2, Q3, Q4, Q6, Q8, Q10, Q11) which are 8 query types out of the 22 TPC-H query types, none of the studied re-optimization algorithms performs better than NoReOpt. Simple query types mean the QEPs for the queries of those query types contain a small number of JOIN operators (usually 2 to 3) and the total number of operators in each of those QEPs is also small (usually 10 to 15). The query response time and the monetary cost for executing the queries of optimization.

As shown in Figures 29 and 30, NoReOpt outperforms the re-optimization algorithms (ReOpt, ReOptML, Sample, ReOptRL and SLAReOptRL) when simple queries are

executed. This is because the main benefits of re-optimization come from the JOIN operator execution but extra overhead is also added. In executing simple queries, the accumulative overheads outweigh the benefits gained from the re-optimizations. In our experiments, the average query response time improvement of the re-optimization algorithms over the baseline algorithm, NoReOpt, for executing JOIN operators is 23% and the average monetary cost is 45%. However, each re-optimization causes around 5% extra query response time and 6% monetary cost on average when simple queries are executed. Those overheads are generated by the additional query processing steps in re-optimizations, such as updating data statistics, running a decision tree model, or neural network. The overheads caused by those procedures are fixed values. When simple queries are executed, the total time and monetary costs are very low, the proportion of the overheads to the execution costs becomes relatively large. Thus, NoReOpt performs the best when simple queries are executed.

On the other hand, when the queries are complex, which means a QEP generated for each of these queries contains a high number of JOIN operators (usually 5 or more) and a high total number of operators (usually over 25), the algorithms with re-optimization (ReOpt, ReOptML, Sample, ReOptRL and SLAReOptRL) outperform the one without re-optimization, NoReOpt. When complex queries are executed, the overall query response time and monetary cost are high and the proportion of the re-optimization overheads in the costs of query execution drops. In our experiments, each re-optimization causes around 2%

extra query response time and 3% monetary cost on average. Since there are more JOIN operators in those types of queries, more benefits are gained from re-optimization. Thus, it is worth applying re-optimization algorithms to those types of queries.

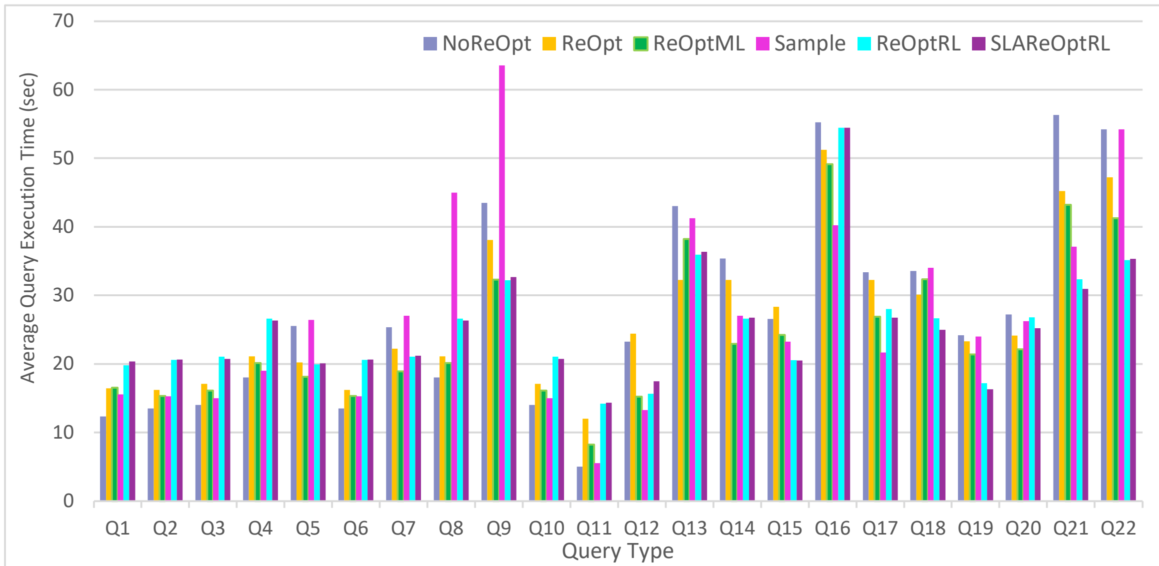


Figure 29. Time performance for executing queries using different algorithms

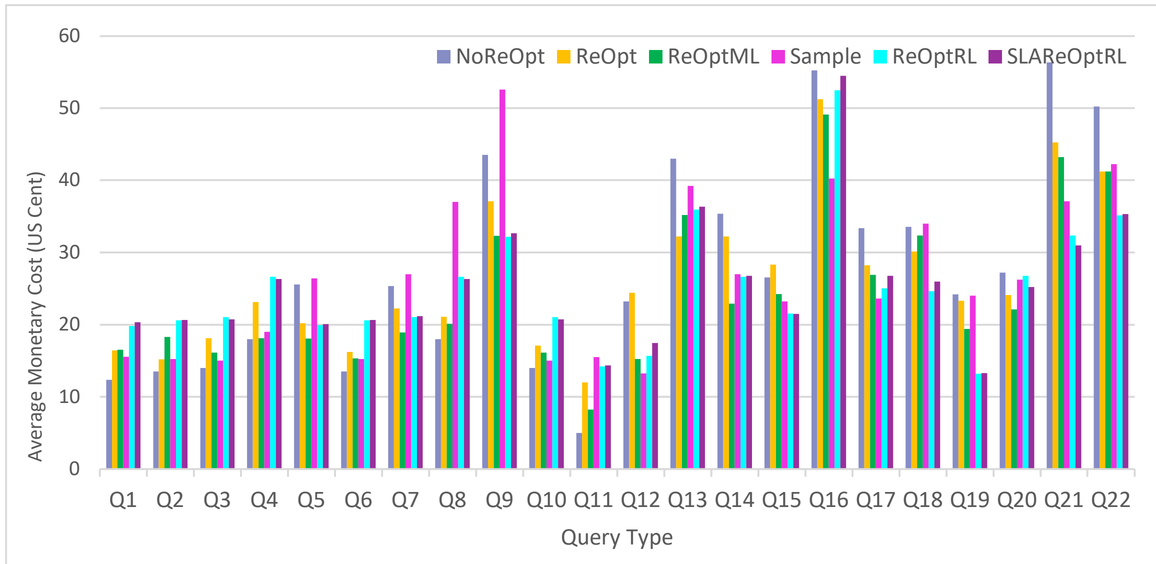


Figure 30. Money cost performance for executing queries using different algorithms

Among the re-optimization algorithms, the above results also show that our proposed algorithms, SLAReOptRL and ReOptRL, still yield less query execution time and monetary costs than the other three algorithms, ReOpt, ReOptML, and Sample. In these experiments, the Reinforcement Learning part of the query processing in our algorithms contributes to these improvements. This is because, in all the three algorithms, ReOpt, ReOptML, and Sample, query re-optimization requires a lot of overhead as the data statistics are required to be accessed and updated frequently, while in our two proposed algorithms that use reinforcement learning, SLAReOptRL and ReOptRL, no data statistics are needed. Instead, our re-optimization is based on the results of learning which is decided quickly. In our experiments, the overhead of the learning decision is only 1.7% of the total query execution time.

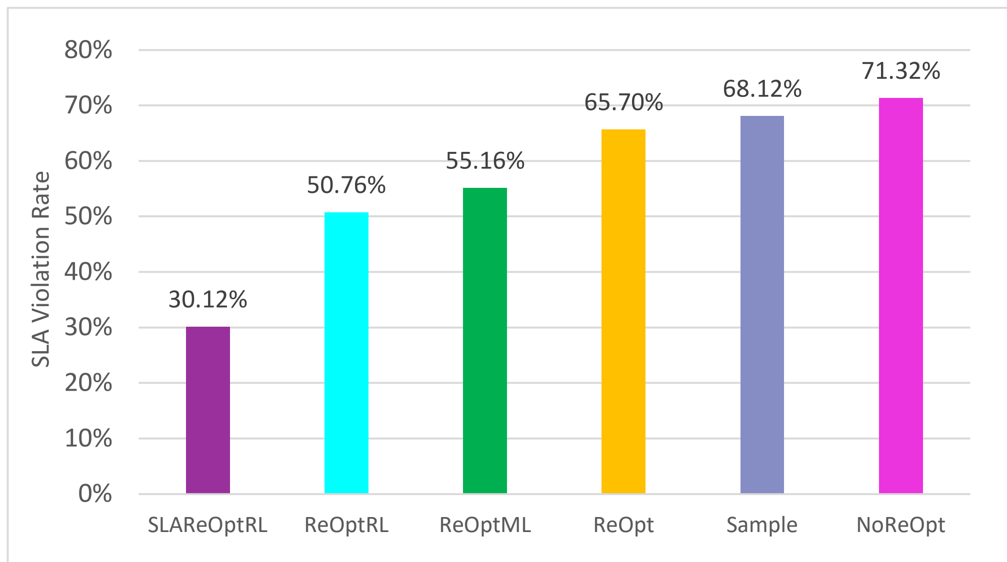


Figure 31. Average SLA violation rate when executing queries using different algorithms

Moreover, from Figure 31, we can also find that by using SLAReOptRL, the SLA violation rate is the lowest one among the SLA violation rates caused by all the algorithms. This shows the positive effect of considering SLA requirements in query re-optimization.

6.2.6.2 Impact of RatioJOIN

In this section, we aim to find out what queries would be suitable for re-optimization. As discussed in Section 6.2.6.1, the major benefits of re-optimization come from optimizing the execution of JOIN operators. However, it is not true that the more JOIN operators a QEP has, the more improvement on query response time will be gained using the re-optimization algorithms. One evidence of this is with the queries of the two query types, Q20 and Q21. Both of these query types have the same number of JOIN operators, but Q21 has more improvement on query response time than Q20 when using our proposed algorithms. The reason is that Q20 has more operators than Q21. Since the re-optimization of a query is conducted after each of the operators in the QEP of the query is executed, there is more overhead caused by re-optimization in Q20 than in Q21 which increases the query response time. Thus, we take both the number of JOIN operators and the total number of operators in a query into consideration when investigating if the query is suitable for re-optimization. Here we study the impacts of RatioJOIN, which is the ratio of the JOIN operators to the total number of operators in a QEP as defined in Equation (10) below.

$$RatioJOIN = \frac{N_j}{N} \qquad (10)$$

where N_j is the number of JOIN operators and N is the total number of operators in a QEP.

Figure 8 shows the relationship between the RatioJOIN and the improvement in query response time when queries of different query types are executed using the re-optimization algorithms and NoReOpt. In this figure, each bar represents the RatioJOIN for each type of query; the yellow bars are for simple query types while the blue bars are for complex query types. Each curve represents the query response time improvement for queries that were executed using a re-optimization algorithm over NoReOpt. For the simple query types, the curves are below zero because NoReOpt outperforms re-optimization algorithms when simple queries are executed. For the complex query types, when the RatioJOIN increases, the improvement of the query response time also increases. We can say that it is more suitable to apply the re-optimization algorithms when queries are complex, i.e., those that have a high number of operators and a high RatioJOIN.

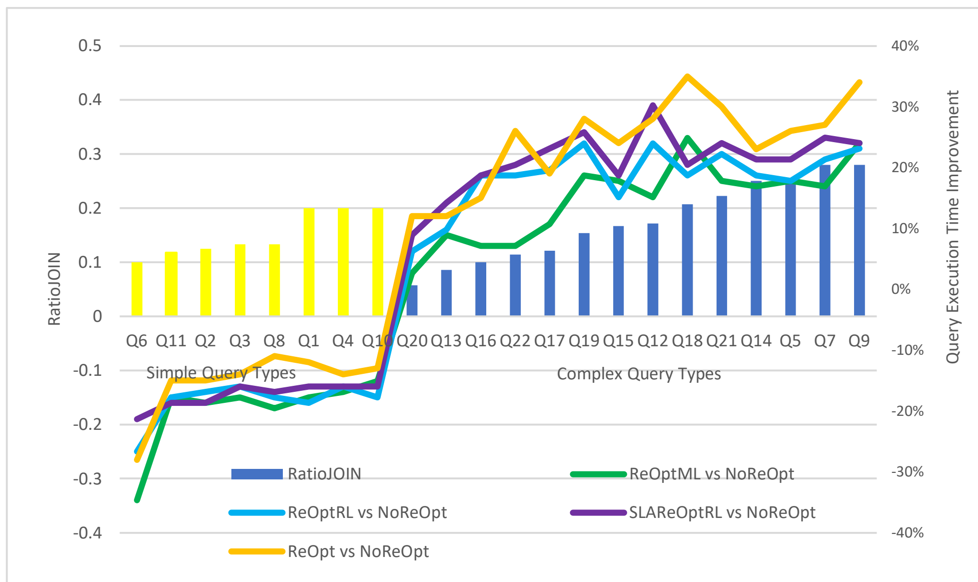


Figure 32. The impacts of RatioJOIN on the query execution time improvement when queries are executed using different re-optimation algorithms compared with NoReOpt

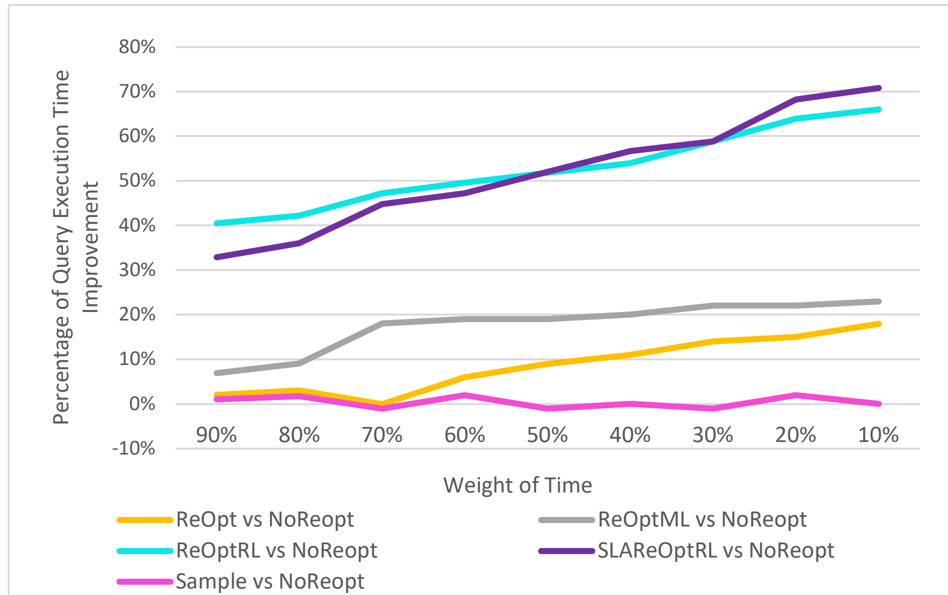
6.2.6.3 Evaluation of ReOptRL and SLAReOptRL without SLA Requirements

In these experiments, we study the performance of our two proposed reinforcement learning based query re-optimization algorithms under scenarios where there are no SLA requirements. For NoReOpt, ReOpt, ReOptML, Sample, and ReOptRL, the results are the same as those presented in Section 6.2.6.1. However, when comparing SLAReOptRL with ReOptRL, the results show that SLAReOptRL spends 5% more time and monetary cost than ReOptRL. This is because, without SLA requirements, both algorithms generate the same QEP for query execution, but in SLAReOptRL, the reward calculation is more complex which incurs more overhead than that in ReOptRL. This leads to a higher query execution time and monetary cost.

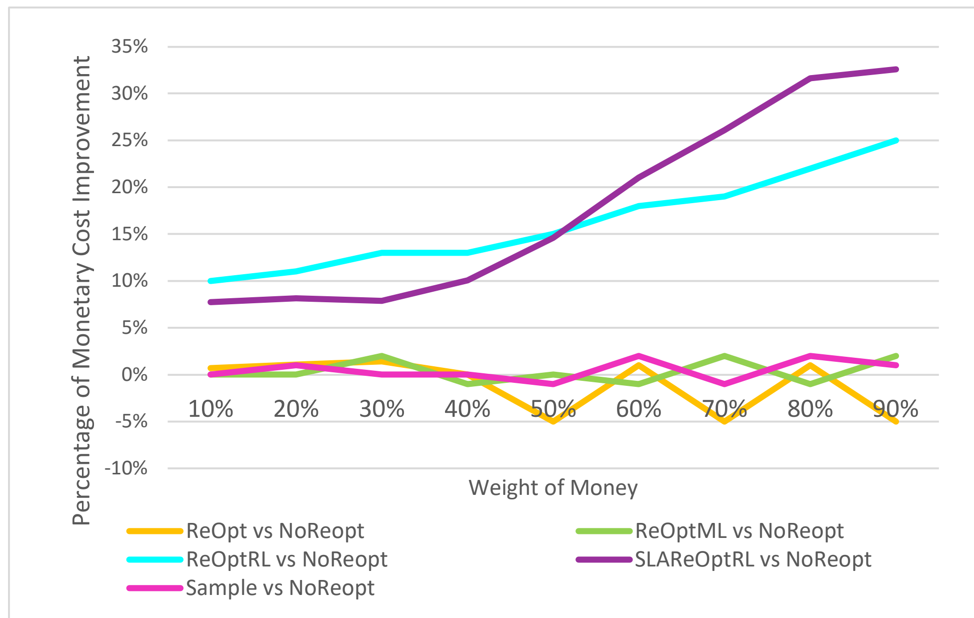
6.2.6.4 Impact of Weights on Different Algorithms

Our algorithms allow users to input their weight profile, and this is also a contribution of our work. This feature is enabled by adjusting the reward function with the weight profile. We want to find out whether our proposed algorithms can adapt the weight profile better than the other competitive algorithms. Figures 33 (a) and (b) show the percentage of improvement of time and monetary cost of each algorithm compared to the baseline NoReOpt on the different weights of query execution time. From the figures, we find that our proposed algorithms have the largest improvement over the baseline. With the increase in weight of time, such improvement also increases. When the weight of time is high at 0.9, our proposed algorithms perform 70% better than the baseline NoReOpt. This happens

because when the weight profile is used in the reward calculation, performing the action of selecting the container that processes the query fast gives the high reward.

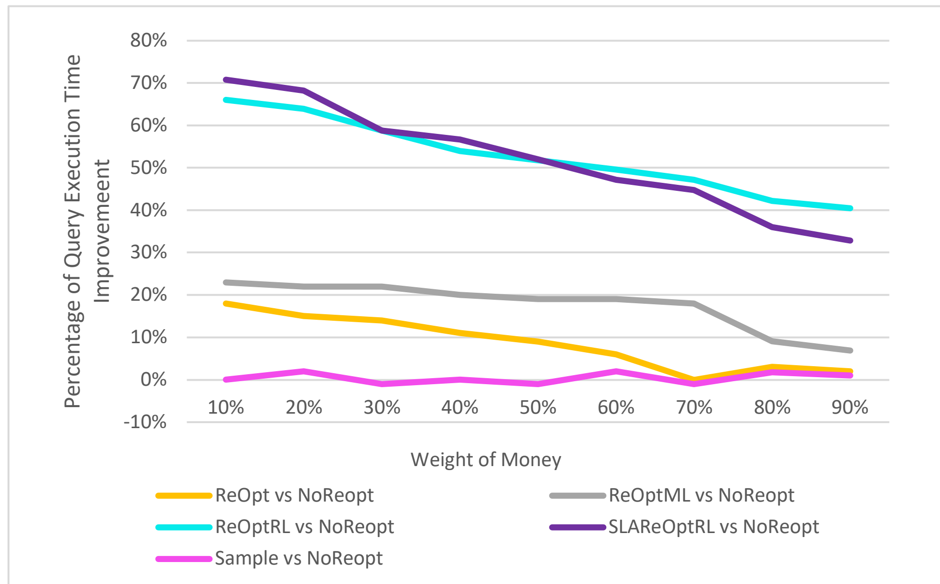


(a)

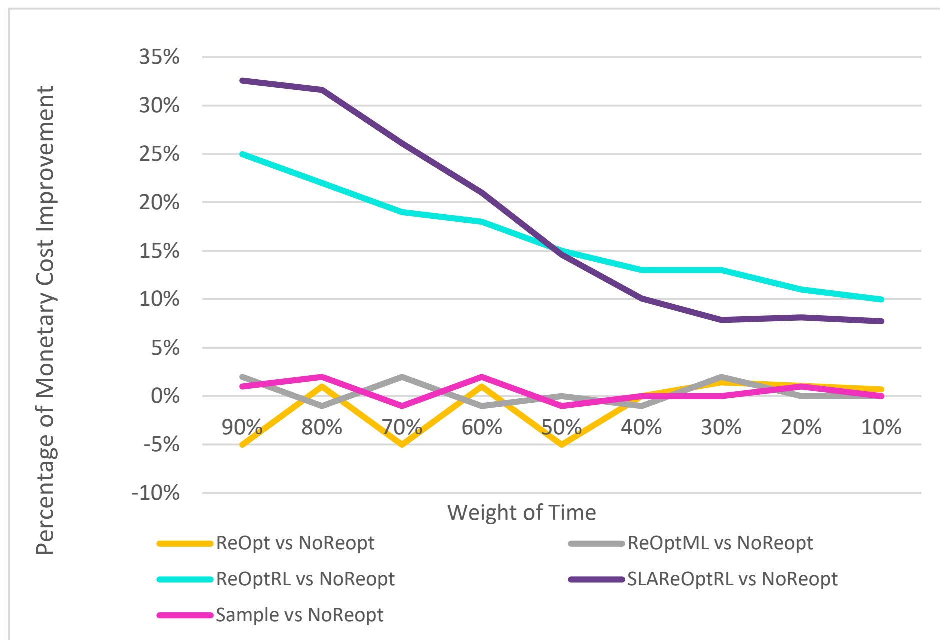


(b)

Figure 33. (a) and (b) Impacts of the weight of time on the performance improvement of the re-optimization algorithms over the baseline algorithm "NoReOpt"



(a)



(b)

Figure 34. (a) and (b) Impacts of the weight of monetary cost on the performance improvement of the re-optimization algorithms over the baseline algorithm "NoReOpt"

Similarly, from the monetary cost perspective, with the increasing weight of time, the improvement of monetary cost decreases, and still, both of our proposed reinforcement

learning-based query re-optimization algorithms perform better than the other four algorithms. Even when the weight of time is high, our algorithms still have the improvement of monetary cost by 10%.

Figures 34 (a) and (b) show the percentages of improvement of time and monetary cost of each algorithm compared to the baseline on the different weights of money. From Figure 34 (a), we find that our two proposed algorithms, ReOptRL and SLAReOptRL, also have the largest improvement over the baseline. When the weight of time is high at 0.9, our proposed algorithms perform 30% better than the baseline NoReOpt. From Figure 34 (b), we find that both of our proposed algorithms perform better than the other four algorithms even when the weight of money is high; they still have the improvement of time cost by 40%. We can conclude that our reinforcement learning-based query re-optimization algorithms are able to reflect the weight profiles on both query execution time and monetary costs. Table 9 gives a summary of the results of the experiments conducted in this algorithm.

6.3 Summary

In this chapter, we presented the correctness proof and the computational complexity and experimental performance evaluations of the proposed algorithms. The time complexity of ReOpt and ReOptML is O (Op^2) and the time complexity of ReOptRL and SLAReOptRL is O ($N_{attr}^2 * Op^2$), where Op is the number of operators in a query execution plan and N_{attr}

is the total number of attributes in all tables in the database. The experimental results show that SLAReOptRL improves query response time (from 12% to 45%) and monetary cost (from 17% to 62%) over ReOptRL, ReOptML, ReOpt, NoReOpt and Sample. Also, SLAReOptRL improves the SLA violation rate from 41% to 20% over those algorithms. The conclusions and future research directions are presented in the next chapter.

Table 9. Performance results (Average Values ± Standard Deviations) of different algorithms. The number (x) after each reported value indicates the ranking of the algorithm with rank (1) being the best

		Avg Query Exe. Time (Sec)	**Avg Monetary Cost (US Cent)**	**SLA Violation Rate**
With SLA	SLAReOptRL	15.38±1.4(1)	6.70±1.1(1)	30.12% (1)
	ReOptRL	17.12±2.3(2)	8.66±1.2(2)	50.76% (2)
	ReOptML	20.12±3.1(3)	13.80±1.6(3)	55.16% (3)
	ReOpt	24.23±2.5(4)	15.30±1.2(4)	65.70% (4)
	Sample	28.13±1.1(5)	17.22±0.9(5)	68.12% (5)
	NoReOpt	28.22 ±3.2(6)	18.23±2.2(6)	71.32% (6)
Without SLA	ReOptRL	17.12±2.3(1)	8.66±1.2(1)	
	SLAReOptRL	18.38±1.4(2)	8.90±1.1(2)	
	ReOptML	20.12±3.1(3)	13.80±1.6(3)	
	ReOpt	24.23±2.5(4)	15.30±1.2(4)	
	Sample	28.13±1.1(5)	17.22±0.9(5)	
	NoReOpt	28.22 ±3.2(6)	18.23±2.2(6)	

CHAPTER VII
CONCLUSIONS AND FUTURE WORK

In this research, four algorithms, ReOpt, ReOptML, ReOptRL, and SLAReOptRL for query re-optimization in cloud database systems are presented.

The first algorithm, ReOpt, re-optimizes a query every time a query operator or a stage of query operators finishes execution. It updates the data statistics and calls the query optimizer to generate a new query execution plan based on the new data statistics. Then the previous query execution plan and the new query execution plan are merged, the query operators that have been executed are eliminated, and then the same process continues for the remaining query operators in the query execution plan that have not been executed. The main characteristic of this algorithm is that it considers both query response time and monetary costs in re-optimization. As the new query execution plan is generated using the new data statistics, it becomes closer to the optimal one. After all the query operators finish execution, the query results are returned to the user.

The second algorithm, ReOptML shares the same re-optimization process as ReOpt, but it does not always re-optimize the query each time a query operator or a stage of query operators finishes execution. Instead, it uses a prediction model based on supervised learning to tell whether a re-optimization should be conducted. This model uses the difference between the old data statistics and the updated data statistics to predict whether

a re-optimization is worth it. As re-optimization is a complex operation in the query process, reducing unnecessary re-optimizations is important to reducing the overhead of the whole query process.

The third algorithm, ReOptRL, uses reinforcement learning instead of supervised learning to optimize the physical query execution plan generated by the existing query optimizer. It uses a logical query execution plan generated by an existing query optimizer. For each query operator in the logical query execution plan, a deep neural network is used to select the optimal physical query operator to execute this logical query operator. This selection is based on a novel reward function that makes use of user preferences on query response time and monetary costs to execute a query and the physical query operator with the lowest cost has a higher chance to be selected again for future queries.

The fourth algorithm, SLAReOptRL, is an extension of ReOptRL. In SLAReOptRL, the re-optimization is based on not only query response time and monetary costs but also the SLA violation rate.

We have analyzed the worst-case time complexity of the four proposed algorithms, ReOpt, ReOptML, ReOptRL and SLAReOptRL. The time complexity of the four proposed algorithms is mainly impacted by the number of operators (Op) and the total number of

attributes in all the database tables (N_{attr}). Besides, we have also proved theoretically that the query results are correct by using the four proposed algorithms.

We have also prorotyped the four proposed algorithms, incorporated them into the open-source DBMS, PostgreSQL, and performed comprehensive experiments evaluating their performance using the TPC-H database benchmark. We have compared ReOpt with NoReOpt in terms of time and monetary costs. Besides, we have compared ReOptML with ReOpt and the existing algorithm, Tukwila, in terms of time and monetary costs. We have also compared the accuracy of re-optimization prediction among different supervised learning models. Finally, we have studied the performance of ReOptRL and its extension SLAReOptRL. We compared this algorithm with the two proposed algorithms, ReOpt and ReOptML, and with the algorithm existing in the literature, Sample. A summary of the experimental results is presented in the following section.

7.1 Summaries of Performance Evaluation Results

In this section, we present the summaries of the experimental performance results of our proposed algorithms.

7.1.1 Summary of Performance Results of ReOpt

Our experimental results show that after query re-optimization, either the query response time or the monetary cost benefits from ReOpt. For executing queries using query re-

optimization on different containers, the query response time is 20% less than the query response time without re-optimization, although the monetary cost before and after re-optimization remains similar, with only a 5% difference. For queries that have operators changed during query re-optimization, the monetary cost is roughly four times less than that without using re-optimization, while the query response time is almost the same in both algorithms.

7.1.2 Summary of Performance Results of ReOptML

ReOptML uses a supervised machine learning-based model to decide whether or not a query should be re-optimized. The experiments conducted show that for skew data, ReOptML improves the query response time (from 13% to 35%) and monetary cost (from 17% to 35%) over the existing algorithms that use either no re-optimization, re-optimization after each stage in the query execution plan is executed, or re-optimization when a checkpoint is reached and the difference between the actual query cost and estimated query cost exceeds some threshold. For uniform data, the proposed algorithm also improves query response time (13% to 21%) over the existing algorithms but does not improve monetary cost.

While our studies have shown that supervised machine learning has positive impacts on deciding whether a re-optimization should be conducted, the supervised machine learning model proposed in this work provides only a binary decision of whether or not a re-

optimization should be carried out, and the model relies on the data statistics which may not be available in all DBMSs.

7.1.3 Summary of Performance Results of ReOptRL and SLAReOptRL

ReOptRL aims to reduce both query response time and monetary costs. SLAReOptRL extends ReOptRL to also consider reducing the amount of SLA requirement violations when re-optimizing queries. The experiments conducted using the TPC-H database benchmark show that both SLAReOptRL and ReOptRL improve query response time (from 12% to 45%) and monetary cost (from 17% to 62%) over the existing algorithms that use either no re-optimization, re-optimization after each operator in the query execution plan (QEP) is executed, supervised machine learning-based query re-optimization, or sample-based re-optimization. In addition, we also find that when there are SLA requirements, SLAReOptRL performs 19% better than ReOptRL on query response time, 20% on query execution monetary costs, and 20% on SLA violation rate. We also find that, when queries are complex, i.e., those queries that have a high total number of operators and a high ratio of JOIN operators to the total number of operators, it is beneficial to apply re-optimization algorithms, especially, our algorithms, ReOptRL and SLAReOptRL, to process queries on cloud database systems.

7.2 Future Research

For future work, we plan to improve our proposed algorithms in the following directions:

a) *Using a reinforcement learning technique to re-optimize the logical plan of a query*

For ReOptRL, the re-optimization requires a logical plan provided by an existing query optimizer. This assumes the existing query optimizer is able to generate an optimal logical plan. In future research, we will generate the optimal logical plan by our algorithm so that the algorithm is independent of any existing query optimizers.

b) Obtaining accurate SLA for each query operator

For SLAReOptRL, when we calculate the reward, the SLA used for each operator is generated by the average overall query response time and monetary cost SLA. However, the different operators should meet different SLA requirements by the characteristics of the operators. In future research, we will study the impact of the SLA requirements on each operator.

c) Improving performance for short queries

For the four proposed algorithms, we have observed from our experimental results that there is a noticeable improvement when executing queries that contain a lot of operators. However, for short queries, i.e., queries that contain only a small number of operators, those algorithms do not perform as well as they perform on long queries. Thus, in future research, we will investigate how to modify our algorithms so that they can improve the performance of short queries.

REFERENCES

[1] N. Bruno, S. Jain and J. Zhou, "Continuous cloud-scale query optimization and processing," VLDB Endow, vol. 6, no. 11, p. 961–972, 2013.

[2] M. Stillger, G. Lohman, V. Markl and M. Kandil, "LEO - DB2's LEarning Optimizer," International Conference on Very Large Data Bases (VLDB '01), p. 19–28, 2001.

[3] W. Wu, J. F. Naughton and H. Singh, "Sampling-Based Query Re-Optimization," International Conference on Management of Data (SIGMOD'16), pp. 1721–1736, 2016.

[4] T. Kaftan, Balazinska, Magdalena, Cheung, Alvin and J. Gehrke, "Cuttlefish: A Lightweight Primitive for Adaptive Query," https://arxiv.org/pdf/1802.09180.pdf, 2018.

[5] I. Trummer, S. Moseley, D. Maram, S. Jo and J. Antonakakis, "SkinnerDB: regret-bounded query evaluation via reinforcement learning," VLDB Endow., vol. 11, no. 12, p. 2074–2077, 2018.

[6] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration," International Workshop on Exploiting Artificial Intelligence Techniques for Data Management (aiDM'18), pp. 1-4, 2018.

[7] C. Costa, L. Cicília and S. António, "Efficient SQL adaptive query processing in cloud databases systems," in IEEE Conference on Evolving and Adaptive Intelligent Systems (EAIS'16), 2016.

[8] Oracle, [Online], Available:"https://docs.oracle.com/en/," [Accessed November 2020].

[9] H. Garcia-Molina, J. D. Ullman and J. Widom, Database Systems: The Complete Book, 2nd edition, Pearson, 2008.

[10] L. Kuiper, "Exploring Query Re-Optimization. In a Modern Database System.," Master Thesis Radboud University in Nijmegen, 2021.

[11] Amazon AWS, [Online], Available: "https://aws.amazon.com/," [Accessed November 2020].

[12] Microsoft Azure, [Online], Available: "https://azure.microsoft.com/en-us/," [Accessed November 2020].

[13] D. Meignan, "A heuristic approach to schedule reoptimization in the context of interactive optimization," Genetic and Evolutionary Computation, 2014.

[14] A. Sebaa and A. Tari, "Query optimization in cloud environments: challenges, taxonomy, and techniques," The Journal of Supercomputing, vol. 75, p. 5420–5450, 2019.

[15] D. Amol, I. Zachary and R. Vijayshankar, "Adaptive Query Processing," Foundations and Trends in Databases, vol. 1, no. 1, pp. 1-140, 2007.

[16] M. Armbrust, K. Curtis, T. Kraska, A. Fox, M. Franklin and D. Patterson, "PIQL: success-tolerant query processing in the cloud," VLDB Endow., vol. 5, no. 3, p. 181–192, 2011.

[17] S. Ewen, K. Holger, V. Markl and V. Raman, "Progressive Query Optimization for Federated Queries," Advances in Database Technology (EDBT'06), vol. 3896, 2006.

[18] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh and M. Cilimdzic, "Robust query processing through progressive optimization.," International conference on Management of data (SIGMOD '04), p. 659–670, 2004.

[19] N. Kabra and D. DeWitt, "Efficient mid-query re-optimization of sub-optimal query execution plans," SIGMOD Rec, vol. 27, no. 2, p. 106–117, 1998.

[20] M. Stonebraker, P. Aoki, R. Devine, W. Litwin and M. Olson, "Mariposa: a new architecture for distributed data," 10th International Conference on Data Engineering (ICDE'94), pp. 54-65, 1994.

[21] C. Wang, Z. Arani, L. Gruenwald and L. d'Orazio, "Adaptive Time, Monetary Cost Aware Query Optimization on Cloud Database Systems," International Conference on Big Data (Big Data), pp. 3374-3382, 2018.

[22] C. Wang, Z. Arani, L. Gruenwald, L. d'Orazio and E. Leal, "Re-optimization for Multi-objective Cloud Database Query Processing using Machine Learning," International Journal of Database Management Systems, vol. 13, no. 1, pp. 21-40, 2021.

[23] C. Wang, L. Gruenwald, L. d'Orazio and E. Leal, "Cloud Query Processing with Reinforcement Learning-Based Multi-objective Re-optimization," International Conference on Model and Data Engineering, pp. 141-155, 2021.

[24] F. Wolf, N. May, P. R. Willems and K.-U. Sattler, "On the Calculation of Optimality Ranges for Relational Query Execution Plans," pp. 663-675, 2018.

[25] L. Nikolay, Z. Kai and Z. Carlo, "Early accurate results for advanced analytics on MapReduce," VLDB Endow., vol. 5, no. 10, p. 1028–1039, 2012.

[26] T. Vodopivec and B. Šter, "Enhancing upper confidence bounds for trees with temporal difference values," IEEE Conference on Computational Intelligence and Games, pp. 1-8, 2014.

[27] B. Kolev, P. Valduriez and C. Bondiombouy, "CloudMdsQL: querying heterogeneous cloud data stores with a common language," Distributed and Parallel Databases, vol. 34, p. 463–503, 2016.

[28] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil and N. Tatbul, "Neo: a learned query optimizer," in VLDB Endow. 12, 11, 2019.

[29] R. Sellami and B. Defude, "Complex Queries Optimization and Evaluation over Relational and NoSQL Data Stores in Cloud Environments," IEEE Transactions on Big Data, vol. 4, no. 2, pp. 217-230, 2018.

[30] K. Herald, S. Eva, M. T. Manolis and Y. Ioannidis, "Schedule optimization for data processing flows on the cloud," International Conference on Management of data (SIGMOD '11), pp. 289-300, 2011.

[31] L. Willis, V. N. Rimma and R. Ian, "Database optimization for the cloud: Where costs, partial results, and consumer choice meet," Conference on Innovative Data Systems Research (CIDR'15), 2015.

[32] S. Thirumuruganathan, S. Hasan, N. Koudas and G. Das, "Approximate Query Processing for Data Exploration using Deep Generative Models," 36th International Conference on Data Engineering (ICDE'20), pp. 1309-1320, 2020.

[33] E. Alpaydin, Introduction to Machine Learning, Third Edition, MIT Press, 2014.

[34] F. Helff, L. Gruenwald and L. d'Orazio, "Weighted Sum Model for Multi-Objective Query Optimization for Mobile-Cloud Database Environments," in EDBT/ICDE Worshops, 2016.

[35] A. Kipf, T. Kipf, B. Radke, V. Leis, P. Boncz and A. Kemper, "Learned Cardinalities: Estimating Correlated Joins with Deep Learning," in 9th Biennial Conference on Innovative Data Systems Research, 2019.

[36] TPC-H Benchmark, [Online], Available: "http://www.tpc.org/tpch/spec/tpch2.8.0.pdf," [Accessed November 2020].

[37] Amazon AWS servicle level agreement, [Online], Available:

"https://aws.amazon.com/legal/service-level-agreements/," [Accessed November 2020].

[38] W. Marco, Reinforcement Learning State-of-the-Art, Springer, 2012.

[39] Y. Park, A. Tajik, M. Cafarella and B. Mozafari, "Database Learning: Toward a Database that Becomes Smarter Every Time," International Conference on Management of Data (SIGMOD '17), p. 587–602, 2017.

[40] J. Ortiz, M. Balazinska, J. Gehrke and S. S. Keerthi, "Learning State Representations for Query Optimization with Deep Reinforcement Learning," the Second Workshop on Data Management for End-To-End Machine Learning (DEEM'18), pp. 1-4, 2018.

[41] M. Lapan, Deep Reinforcement Learning Hands-On: Apply modern RL methods, with deep Q-networks, value iteration, policy gradients, TRPO, AlphaGo Zero and more, Packt Publishing, 2018.

[42] M. Ramicic and A. Bonarini, "Attention-Based Experience Replay in Deep Q-Learning," the 9th International Conference on Machine Learning and Computing (ICMLC'17), p. 476–481, 2017.

[43] S. Gupta, Express Learning - Database Management Systems, Pearson India, 2012.

[44] PostgreSQL, "https://www.postgresql.org/docs/," [Online].

[45] D. Tansel, A. B. Murat and C. Ahmet, "Robust heuristic algorithms for exploiting the common tasks of relational cloud database queries," Appl. Soft Comput, vol. 30, no. C, p. 72–82, 2015.

[46] I. Trummer and C. Koch, "Multi-objective parametric query optimization," VLDB Endow, vol. 8, no. 3, p. 221–232, 2014.

[47] C. Wu, J. Alekh, A. Saeed, P. Hiren, L. Wangchao, Q. Shi and R. Sriram, "Towards a learning optimizer for shared clouds," in VLDB Endow. 12, 3, November,2018.

[48] Z. Karampaglis, A. Gounari and Y. Manolopoulos, "A Bi-objective Cost Model for Database Queries in a Multi-cloud Environment," the 6th International Conference on Management of Emergent Digital EcoSystems (MEDES '14)., p. 109–116, 2014.

[49] N. Papailiou, I. Konstantinou, D. Tsoumakos, P. Karras and N. Koziris, "H2RDF+: High-performance distributed joins over large-scale RDF graphs," IEEE International Conference on Big Data, 2013.

[50] Y. Silva, P.-A. Larson and J. ZhouZhou, "Exploiting Common Subexpressions for Cloud Query Processing," 28th International Conference on Data Engineering, pp. 1337-1348, 2012.