Cloud Query Processing with Reinforcement Learning-based Multi-Objective Re-optimization

Chenxiao Wang, Le Gruenwald, Laurent d'Orazio, Eleazar Leal

Cloud Query Processing with Reinforcement Learning-based Multi-Objective Re-optimization

Chenxiao Wang[1], Le Gruenwald[1], Laurent d'Orazio[2], and Eleazar Leal[3]

[1] University of Oklahoma, Norman, Oklahoma, USA
{chenxiao, ggruenwald}@ou.edu
[2] CNRS IRISA, Rennes 1, University, Lannion, France
laurent.dorazio@univ-rennes1.fr
[3] University of Minnesota Duluth, Duluth, Minnesota, USA
eleal@d.umn.edu

Abstract. Query processing on cloud database systems is a challenging problem due to the dynamic cloud environment. The configuration and utilization of the distributed hardware used to process queries change continuously. A query optimizer aims to generate query execution plans (QEPs) that are optimal meet user requirements. In order to achieve such QEPs under dynamic environments, performing query re-optimizations during query execution has been proposed in the literature. In cloud database systems, besides query execution time, users also consider the monetary cost to be paid to the cloud provider for executing queries. Thus, such query re-optimizations are multi-objective optimizations which take both time and monetary costs into consideration. However, traditional re-optimization requires accurate cost estimations, and obtaining these estimations adds overhead to the system, and thus causes negative impacts on query performance. To fill this gap, in this paper, we introduce ReOptRL, a novel query processing algorithm based on deep reinforcement learning. It bootstraps a QEP generated by an existing query optimizer and dynamically changes the QEP during the query execution. It also keeps learning from incoming queries to build a more accurate optimization model. In this algorithm, the QEP of a query is adjusted based on the recent performance of the same query so that the algorithm does not rely on cost estimations. Our experiments show that the proposed algorithm performs better than existing query optimization algorithms in terms of query execution time and query execution monetary costs..

Keywords: Query Optimization· Cloud Databases · Reinforcement Learning · Query Re-optimization.

1 Introduction

In a traditional database management system (DBMS), finding the query execution plan (QEP) with the best query execution time among those QEPs generated by a query optimizer is the key to the performance of a query. In a

cloud database system, minimizing query response time is not the only goal of query optimization. As hardware usages are charged on-demand and scalability is available to users, monetary cost also needs to be considered as one of the objectives for optimizing QEPs in addition to query response time. In order to do that, the query optimizer evaluates the time and monetary costs of different QEPs in order to derive the best QEP for a query. 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 [4,13]. These methods are heuristic-based and the adjustment of QEP is not adaptable to the dynamic environment. More recently, machine learning-based algorithms are introduced [9,20]. More accurate cost estimations are made by the data statistics estimated by the machine learning models. The optimizer uses these cost estimations to adjust the QEP. Again, even those methods improved 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. To eliminate this problem, in this paper we propose an algorithm, called Re-OptRL, for query re-optimization that makes use of reinforcement learning (RL) to find the best QEP without relying on data statistics.

RL is about taking suitable actions to maximize reward in a particular situation. The decision of choosing actions follows the trial-and-error method and is evaluated by the reward. The learning model is adjusted with the reward after each action has been performed. We choose RL instead of supervised learning methods [10] because unlike supervised learning, RL does not require a labeled dataset of past actions to be available to train the learning model in advance before it can be used to predict future actions. Query optimization in the database system can be regarded as a series of actions and the best actions can be learned from evaluating the historical optimizations. In the proposed algorithm ReOptRL, the QEP of a query is adjusted during the query execution independently of data statistics. The adjustment is based on the performance of historical queries that are executed on the system recently. Evaluating these historical executions is reasonable especially on the cloud database system. As a large number of different queries are executed frequently, for the same query, the time gap between its incoming query and its historical query is short. The algorithm monitors and keeps learning from these executions while more incoming queries are executed. Some algorithms used reinforcement learning in adjusting their QEPs also, but these adjustments are only focusing on the JOIN order of queries [5,8]. Our algorithm extends these features to not only adjust the QEP itself but also to find the best allocations to execute different operators in the query, i.e, to find which

machines should be used to execute which operator. In a cloud database system, each operator can be executed on a different machine, and different machines can have different hardware configurations with different usage prices. This means that for the same query operator, not only different execution times but also different monetary costs can incur depending on which machine is used to execute the operator. An operator executed on machine A may have a lower execution time but may cost more money, on the other hand, the same operator executed on machine B may have a higher execution time but may cost less money. Those differences in execution times and monetary costs for individual operators eventually affect the overall performance in accumulation for the entire query. Our goal is to fulfill both user's execution time and monetary cost requirements of the overall query performance. Due to this reason, optimal allocations of operator executions to appropriate machines on the cloud are as important as optimal QEPs themselves. Our contributions in this work are the following:

- We propose an algorithm that uses RL to perform multi-objective query processing for an end-to-end cloud database system.
- Our algorithm employs a new reward function designed specifically for query re-optimization.
- We present comprehensive experimental performance evaluations. The experimental results show that our algorithm improves both query execution time and monetary costs when comparing to existing query optimization algorithms.

The remaining of this paper is organized as follows: Section 2 discusses the related work; Section 3 presents the proposed reinforcement learning-based query re-optimization algorithm; Section 4 discusses the experimental performance evaluations; and finally, Section 5 presents the conclusions and discusses future research directions.

2 Related Work

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. Also, sometimes, a human-in-the-loop was needed to analyze and adjust these heuristics [5]. These add additional overheads to the overall performance of queries. Unfortunately, these heuristic solutions can often miss good execution plans. More importantly, traditional query optimizers rely on static strategies, and hence do not learn from previous experiences. Traditional systems plan a query, execute the query plan, and forget that 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.

Machine learning techniques have been used recently in query optimizations for different purposes. In earlier works, Leo [15] learns from the feedback of

executing past queries by adjusting its cardinality estimations over time, but this algorithm requires human-tuned heuristics, and still, it only adjusts the cardinality estimation model for selecting the best join order in a query. More recently, the work in [20] 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. In this work, only join orders, not the entire query, are optimized. In [18], the authors examine the use of deep learning techniques in database research. Since then, reinforcement learning is also used. The work proposes a deep learning approach for cardinality estimation that is specifically designed to capture join-crossing correlations. SkinnerDB [16] is another work that uses a regret-bounded reinforcement learning algorithm to adjust the join order during query execution. None of these machine learning-based query optimization algorithms is designed for predicting the action that should follow after one query operator has been executed. Recently, there have been several exciting proposals in putting reinforcement learning (RL) inside a query optimizer. As described in [19] and shown in Fig.1, 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 performs an action from 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 r_t 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.

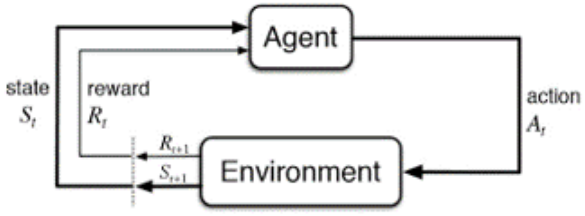

Fig. 1: General Procedure of Reinforcement Learning [19]

Ortiz et al. [12] apply deep RL to learn a representation of queries, which can then be used in downstream query optimization tasks. Liu [6] and Kipf [5] use DNNs to learn cardinality estimates. Closer to our work is Marcus et al.'s work of a deep RL-based join optimizer, ReJOIN [8], which orders a preliminary view of the potential for deep RL in this context. The early results reported in [8] give a 20% improvement in the query execution time of Postgres. However, they have evaluated only 10 out of the 113 JOB queries.

3 Deep Reinforcement Learning based Query Re-Optimization

In this section, we present our query re-optimization algorithm that makes use of reinforcement learning, ReOptRL. First, we discuss our query processing procedure in the cloud database system. Then we describe how the deep reinforcement learning algorithm is used in our query processing to select the best actions for the performance of queries. In this algorithm, a query will be converted into a logical plan by a traditional query parser. Then for each logical query operator, we use a deep reinforcement learning model to select what physical operator and which machines (containers) should be used to execute this logical operator so that each operator execution is optimized to gain the maximum improvement on overall performance. These selections are learned from the same operator executed previously in the system. Besides that, as in large applications, there will be a large number of queries running frequently. For a query operator, it is reasonable to utilize the performance of its previous executions of the same query to predict the performance of its current execution because the time between the two executions is short.

3.1 The Environment of Our Cloud Database System

In this section, we first briefly describe our environment for query processing. Our applications mainly focus on processing queries in a mobile cloud environment. Fig. 2 shows the process flow of query processing in the mobile-cloud database system that we have developed [17]. In this architecture, the mobile device is for the user to access the database and input queries, the data owner is a server onsite that contains private data, and the cloud provider owns the cloud database system. Executing a query incurs three different costs: the monetary cost of query execution on the cloud, the overall query execution time, and the energy consumption on the mobile device where the query might be executed. These three costs constitute the multi-objectives that the query optimizer needs to minimize to choose the optimal query execution plan (QEP). Different QEPs are available due to the elasticity of the cloud which considers multiple nodes with different specifications. In this paper, we focus on query processing on the cloud provider's part. We consider both the query execution time and monetary cost, but not the energy cost on mobile devices.

Different users have different preferences for choosing a suitable QEP for their purposes. In an application scenario where many queries are executed per day, organizations may want to minimize the monetary cost spent for query execution to fit their budget. They may also want to minimize query execution time to meet customers' query response time requirements and to optimize employees' working time. In order to incorporate these user's preferences into the query optimization algorithm, we use the Normalized Weighted Sum Model we have developed in [3] to select the best plan. In this model, every possible QEP alternative is rated by a score that combines both the objectives, query response time, and monetary cost, with the weights defined by the user and the environment for each

objective, and the user-defined acceptable maximum value for each objective. 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 to 1 which defines the preferences. The following function is used to compute the score of a QEP:

$$A_i^{WSM-SCORE} = \sum_{j=1}^{n} W_j \frac{a_{ij}}{m_j} \tag{1}$$

where a_{ij} is the value of QEP 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 the user and environment weights for objective j, which is defined as follows:

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

where uw_j and ew_j describe the user and the environmental weight for objective j, respectively. These weights are user-defined. Since the different objectives are representative of different costs, the model chooses the alternative with the lowest score to minimize the costs.

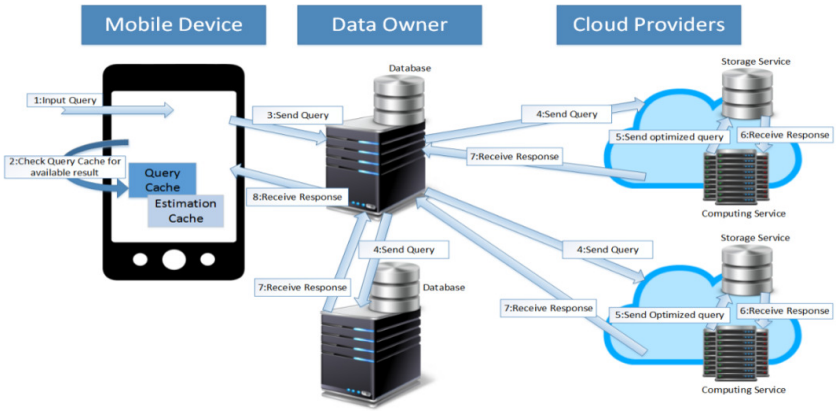

Fig. 2: Mobile Cloud Database Environment

3.2 Overview of Reinforcement Learning based Query Processing Algorithm (ReOptRL)

In this work, we use a policy gradient deep RL algorithm 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 plan parsed from a traditional query optimizer.

The input of the neural network is the vector of 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 execution container 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 will be explained in Section 3.3. The Weights of the neural network is updated after several episodes using existing techniques, such as back-propagation [6].

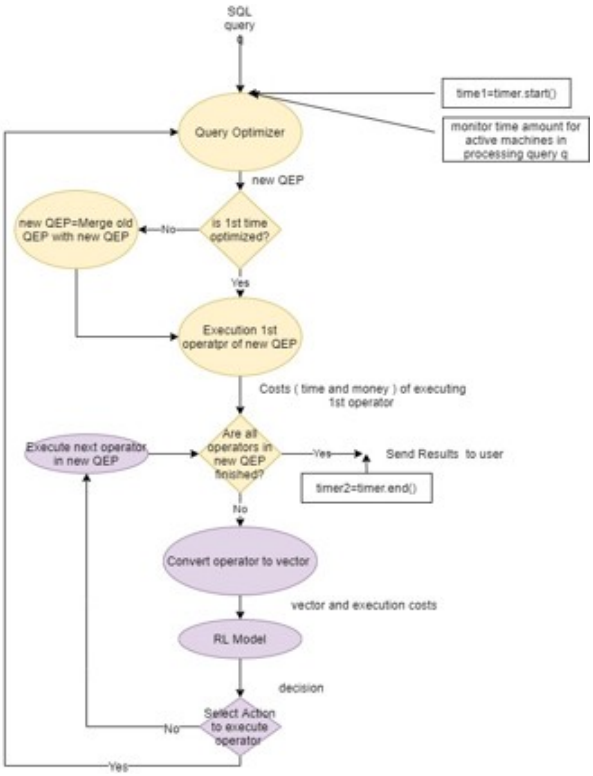

Fig. 3: Procedure of the proposed algorithm, ReOptRL

Fig.3 shows the major steps in query processing when ReOptRL is incorporated for query re-optimization. First, the optimizer receives a query and then the query is compiled into a QEP (logical plan). 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 physical operator and a container to execute this operator. Then this operator is sent to execution and the execution time and monetary costs of this execution are recorded to update the reward function. Once the reward function is updated, the weights of this RL model are adjusted according to the updated reward function. Then the updated RL model is ready for future action selections of the same operator.

There are various kinds of reinforcement learning algorithms. Q-Learning is one of the popular value-based reinforcement learning algorithms [19]. 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 [11].

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

In Equation (3), $Q(S_t, a_t)$ is an evaluated value (called Q-value) of 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 γ 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 γ controls the weight of future rewards. If $\gamma = 0$, the agent only cares for the first reward, and if $\gamma = 1$, the agent cares for all the rewards in the future [5]. R is the reward which is described in Section 3.3

Fig.4 shows the pseudo-code of the proposed algorithm. First, the optimizer receives a query and then the query is compiled into a QEP (logical plan) (Line 4). Then, the QEP is converted into a vector representation. By doing this, the current QEP represents the current state and can be used as the input of a neural network. We use a one-hot vector to present this vector and this technique is adapted from the recent work [7] (Line 5). This vector is sent to the RL model, which is a neural network as described in Section 3.1. The RL model will evaluate the Q-values for all the potential actions to execute the next available query operator (Line 6). Each of these actions includes two aspects, the optimal physical operator and the best container to execute the next available query operator. Then the action with the best Q-value will be selected and performed by the DBMS (Line 7). After that, the executed operator is discarded from the QEP and the time and monetary costs to execute this operator are used to compute the reward for this action (Line 8). The reward is updated with the time and monetary cost to execute the operator and then the expected Q-value is updated by the Bellman Equation (3) with the updated reward (Line 9-11). The weights of the neural network are updated accordingly by the back-propagation method (Line 12). 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 15).

Algorithm 1: Query Processing with Reinforcement Learning Based Re-Optimization (Re-OptRL)

Data: SQL query, Weight Profile wp, Reward Function R (), Learning rate α, Discount rate γ
Result: The query result set of the input query

```
1  t=0;
2  Result = ∅;
3  Q_t= 0;
4  while QEP ≠ ∅ do
5  |  State S_t = convert QEP to a state vector;
6  |  Actiont=RunLearningModel (S_t, wp);
7  |  Result=Result ∪ execute (Op, Action_t);
8  |  QEP=QEP-Op;
9  |  Update R_t=R (wp, Actiont.time, Actiont.money));
10 |  Obtain Q-value of next state Q_{t+1} from the neural network;
11 |  Update Q-value of current state Q_t = Bellman(Q_t, Q_{t+1}, R_t, α, γ);
12 |  Update Weights in the neural network;
13 |  t=t+1;
14 end
15 return Result;
```

Fig. 4: Pseudo code of Proposed algorithm (ReOptRL)

3.3 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. In our algorithm, we would like the actions with low query execution time and monetary cost to be more likely chosen. To reflect this feature, here we define the reward function as follows:

$$Reward\ R = \frac{1}{1 + (W_t * T^q_{op}) + (W_m * M^q_{op})} \tag{4}$$

where W_t and W_m are the time and monetary weights provided by the user, T^q_{op} and M^q_{op} is the time and monetary costs for executing the current operator op in query q. According to this reward function, the query is executed based on the user's preference which is either the user wants 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 to 1. 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, the weight profile for this user probably would be $< W_t = 0.9, W_m = 0.1 >$. The detail can be found in our previous work [3]. This reward function is a monotonic decreasing function. With the increase of $(W_t * T^q_{op}) + (W_m * M^q_{op})$, which is the total costs of executing a query operator, the reward decreases. Notice that, as $(W_t * T^q_{op}) + (W_m * M^q_{op})$ 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 $1 + (W_t * T^q_{op}) + (W_m * M^q_{op})$ 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.

4 Performance Evaluation

In this section, we first describe the hardware configuration, database benchmark, and parameters we used in our experimental model. We also introduce the algorithms we compared with our algorithm.

4.1 Hardware Configuration, Database Benchmark, and Parameter Values

There are two sets of machines 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, called small containers, have one Intel Xeon E5-2682 processor running at 2.5GHz with 1 GB of DRAM. The other 5 VPSs, called large containers, each 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 [14]. The data are distributed among these VPSs. The queries and database tables are generated using the TPC-H benchmark [1]. There are eight database tables with a total size of 1,000 GB. 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 TPC-H benchmark [1]. In the experiments, to update the Q-value using the Bellman equation as shown in Equation (3) discussed in Section 3.2, we set the learning rate α as 0.1 and discount rate γ as 0.5.

4.2 Evaluation of ReOptRL

In this section, we compare the query processing performances obtained when the following query re-optimization algorithms are incorporated into query processing: 1) our proposed algorithm (denoted as **ReOptRL**); 2) the algorithm where a query re-optimization is conducted automatically after the execution of each stage in the query is completed (denoted as **ReOpt**), which we developed based on the state-of-art works [17,2]; 3) the algorithm where a query re-optimization is conducted by a supervised machine learning model decision (denoted as **ReOptML**); 4) the query re-optimization algorithm which uses sampling-based query estimation (denoted as **Sample**) proposed in [21]; and 5) the query processing algorithm that uses no re-optimization (denoted as **NoReOpt**).

We use NoReOpt as the baseline and the other algorithms are compared to the baseline. The two figures, Fig. 5 and Fig. 6, show the improvement of each algorithm over the baseline. From Fig 5, we can see that, from the query execution time perspective, ReOptRL performs 39% better on average than NoReOpt, while ReOptML performs 27%, ReOpt 13% and Sample 1% better than NoReOpt. There are 15 out of 22 query types that have better performance if our algorithm is used. The best case is query 9 (Q9), for which our algorithms perform 50% better than NoReOpt. From Fig 6, from the monetary cost perspective, query processing uses ReOptRL performs 52% better on average than NoReOpt, while ReOptML performs 27%, ReOpt 17%, and Sample 5% better than NoReOpt. There are 14 out of 22 query types that have better performance if our proposed algorithm is used. The best case is also Q9, for which our algorithms perform 56% better than NoReOpt.

The above results show that overall our proposed algorithm improves more time and monetary cost than the four algorithms, ReOpt, ReOptML, Sample, and NoReOpt. Especially, the monetary cost has a significant improvement (56% better than NoReOpt); however, for simple query types (1,2,3,4,6,8,10,11) which are 8 query types out of 22 TPC-H query types, our algorithm does not improve their performance due to the overhead involved.

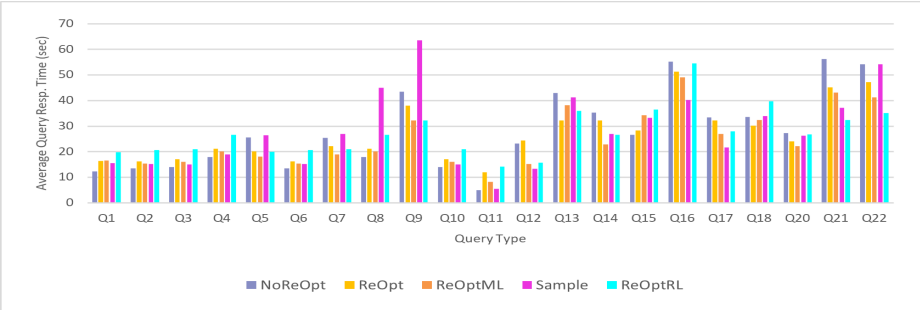

Fig. 5: Time cost performance of executing query using different algorithms

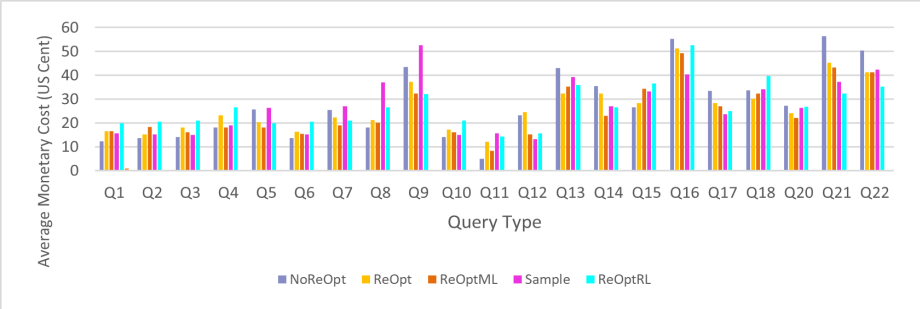

Fig. 6: Monetary cost performance of executing queries using different algorithms

In these experiments, the Reinforcement Learning part of the query processing contributes to these improvements and it is beneficial in the following aspect. In all the three algorithms, ReOpt, ReOptML, and Sample, query re-optimization requires a lot of overhead data statistics that need to be accessed and updated frequently. In our proposed algorithms, no data statistics are needed, and re-optimization is based on the results of learning which can be decided quickly.

5 Impacts of Weight Profiles

Our algorithm allows users to input their weight profiles and this feature is reflected in the reward function. We study how our proposed algorithm works under different weight profiles submitted by users. Fig.7 shows the performance of the monetary cost of each algorithm and the percentage of improvement of the monetary cost of each algorithm compared to the baseline on different weights. Notice that the Weight of Money in our application is (1-Weight of Time).

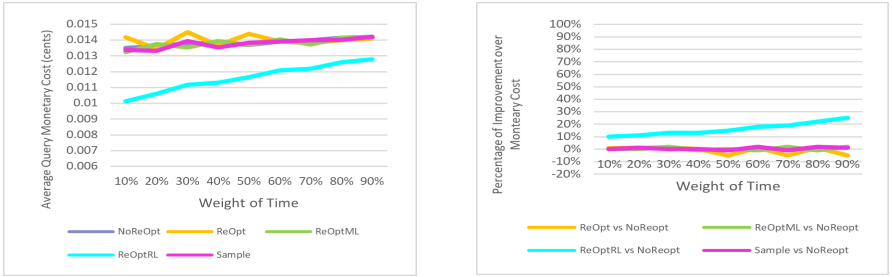

Fig. 7: Impacts of the weight of query execution time on the performance improvement of the algorithm over the baseline algorithm, NoReopt

From Fig.7, we can see that, when the weight of money increases, the monetary cost to execute the query decreases accordingly. Even when the weight of money is low, our algorithm still outperforms other algorithms on the monetary cost (by 10%). This happens because by using the learning model, our algorithm always chooses the right physical operator and container to execute the operator. Similarly, from the monetary cost perspective, our algorithm performs better on average than the other three algorithms.

The above results show that our algorithm can adapt to the user's weight profile very well. When the user has a high demand on either query execution time or money cost, our algorithm still outperforms the other algorithms.

6 Conclusion and Future Work

This paper presents an algorithm called ReOptRL for processing queries in a cloud database system taking both query execution time and money costs to be paid to the cloud provider into consideration. The algorithm uses a reinforcement learning-based model to decide the physical operator and machines to execute an operator from a query execution plan (QEP) of a query. The experimental performance evaluations using the TPC-H benchmark show that our proposed algorithm, ReOptRL, improves the query response time (from 12% to 39%) and monetary cost (from 17% to 56%) over the existing algorithms that use either no re-optimization, re-optimization after each stage in the query execution plan (QEP) is executed, supervised machine learning-based query re-optimization, or sample-based re-optimization.

While our studies have shown that reinforcement learning has positive impacts on the optimization and execution of each operator in a query, our algorithm currently does not consider the Service Agreement Level (SLA), which is an important feature of cloud database systems. For future work, we will investigate techniques to incorporate this feature into our algorithm by adjusting the reward function to consider SLA.

Acknowledgement

This work is partially supported by the National Science Foundation Award No. 1349285.

References

1. Barata, M., Bernardino, J., Furtado, P.: An overview of decision support benchmarks: Tpc-ds, tpc-h and ssb. In: Rocha, A., Correia, A.M., Costanzo, S., Reis, L.P. (eds.) New Contributions in Information Systems and Technologies. pp. 619–628 (2015)
2. Bruno, N., Jain, S., Zhou, J.: Continuous cloud-scale query optimization and processing. Proc. VLDB Endow. **6**(11), 961–972 (Aug 2013)

3. Helff, F., Gruenwald, L., d'Orazio, L.: Weighted sum model for multi-objective query optimization for mobile-cloud database environments. In: Proceedings of the Workshops of the EDBT/ICDT 2016 Joint Conference. vol. 1558 (2016)
4. Kabra, N., DeWitt, D.J.: Efficient mid-query re-optimization of sub-optimal query execution plans. p. 106–117. SIGMOD '98 (1998)
5. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P.A., Kemper, A.: Learned cardinalities: Estimating correlated joins with deep learning. In: 9th Biennial Conference on Innovative Data Systems Research, CIDR 2019 (2019)
6. Liu, H., Xu, M., Yu, Z., Corvinelli, V., Zuzarte, C.: Cardinality estimation using neural networks. p. 53–59. CASCON '15, IBM Corp. (2015)
7. Marcus, R., Negi, P., Mao, H., Zhang, C., Alizadeh, M., Kraska, T., Papaemmanouil, O., Tatbul, N.: Neo: A learned query optimizer **12**(11), 1705–1718 (2019)
8. Marcus, R., Papaemmanouil, O.: Deep reinforcement learning for join order enumeration. aiDM'18 (2018)
9. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data. p. 659–670. SIGMOD '04 (2004)
10. Murphy, K.P.: Machine Learning: A Probabilistic Perspective. The MIT Press (2012)
11. Ohnishi, S., Uchibe, E., Yamaguchi, Y., Nakanishi, K., Yasui, Y., Ishii, S.: Constrained deep q-learning gradually approaching ordinary q-learning. Frontiers in Neurorobotics **13**, 103 (2019)
12. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: Learning state representations for query optimization with deep reinforcement learning. In: Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning. DEEM'18 (2018)
13. Park, Y., Tajik, A.S., Cafarella, M., Mozafari, B.: Database learning: Toward a database that becomes smarter every time. In: Proceedings of the 2017 ACM International Conference on Management of Data. p. 587–602. SIGMOD '17 (2017)
14. PostgreSQL: (2021), https://www.postgresql.org/
15. Stillger, M., Lohman, G.M., Markl, V., Kandil, M.: Leo - db2's learning optimizer. In: Proceedings of the 27th International Conference on Very Large Data Bases. p. 19–28. VLDB '01 (2001)
16. Trummer, I., Moseley, S., Maram, D., Jo, S., Antonakakis, J.: Skinnerdb: Regret-bounded query evaluation via reinforcement learning. Proc. VLDB Endow. **11**(12), 2074–2077 (Aug 2018)
17. Wang, C., Arani, Z., Gruenwald, L., d'Orazio, L.: Adaptive time, monetary cost aware query optimization on cloud database systems. In: IEEE International Conference on Big Data, Big Data 2018. pp. 3374–3382. IEEE (2018)
18. Wang, W., Zhang, M., Chen, G., Jagadish, H.V., Ooi, B.C., Tan, K.L.: Database meets deep learning: Challenges and opportunities. SIGMOD Rec. **45**(2), 17–22 (Sep 2016)
19. Wiering, M., van Otterlo, M.: Reinforcement Learning: State-of-the-Art. Springer Publishing Company, Incorporated (2014)
20. Wu, C., Jindal, A., Amizadeh, S., Patel, H., Le, W., Qiao, S., Rao, S.: Towards a learning optimizer for shared clouds. Proc. VLDB Endow. **12**(3), 210–222 (Nov 2018)
21. Wu, W., Naughton, J.F., Singh, H.: Sampling-based query re-optimization. p. 1721–1736. SIGMOD '16, Association for Computing Machinery (2016)