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

Chenxiao Wang, Zach Arani, Le Gruenwald
School of Computer Science
University of Oklahoma
Norman, Oklahoma, USA
{chenxiao, myrrhman, ggruenwald}@ou.edu

Laurent d'Orazio
CNRS IRISA
Rennes 1 University
Lannion, France
laurent.dorazio@univ-rennes1.fr

Eleazar Leal
Department of Computer Science
University of Minnesota Duluth
Duluth, Minnesota, USA
eleal@d.umn.edu

***Abstract*—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 (QEP). In order to optimize a query with a more accurate cost estimation, 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 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 paper, we present a query processing algorithm using 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. The experiments comparing our proposed algorithm with existing query processing algorithms show that using machine learning based re-optimization does improve query response time no matter queries are executed on uniform or skew data; and 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.**

Keywords—Query Optimization, Cloud Databases, Machine Learning, Query Re-optimization

I. Introduction

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 [1, 2]. Thus, query execution plans (QEPs) 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 [3]. This strategy consists in 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 in between each stage. 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 [1, 4]. Operators that do not rely on the completion of others are grouped together and such groups are called "Stages". For example, if a query plan 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 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 multiples 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 our previous work [5], we presented a query processing algorithm that performs query re-optimization after the completion of each stage. However, our previous 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 Fig. 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 of running this query. The details on these findings are reported in Section III.

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 paper 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 in order to learn to predict the effectiveness of query re-optimizations, and this is done with the purpose of helping the query optimizer avoid unnecessary query re-optimizations for the future queries.

```
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_i
```

Fig. 1 Query 1

While machine learning has been used to improve query processing in a number of recent works, such as [6, 7], to the best of our knowledge, it has 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 in the many features that influence query cost estimations, such as selectivity, cardinality, min and max values of a column, 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 in the large space of possible machine learning models. Supervised learning algorithms like Decision Tree [8] and Support Vector Machine [9] are suitable but need to be used in a correct manner. The third issue is about the collection of the historical data on the selected subset of features that is needed to train the prediction model constructed using the selected machine learning algorithm. The fourth issue consists in measuring the effectiveness of the learning algorithm. Some works such as [10] show the learning algorithm is effective for their own purposes, such as improving the cost estimation, but actually, not all of them are effective in actual query execution performance. Our proposed technique addresses all these issues.

In this paper we make the following contributions:

- We present a novel query processing algorithm for cloud databases that uses machine learning-based re-optimization to optimize query response time and monetary costs. We discuss the feature selection, training data collection, machine learning model selection, and integration of the selected machine learning model into query processing.

- We present a comprehensive experimental study evaluating the accuracy of different machine learning models, the query response time and monetary cost of the proposed query processing algorithm when operating under different machine learning models, and comparing the proposed query processing algorithm with existing query processing algorithms.

The remaining of this paper is organized as follows: Section II discusses the related work; Section III provides the details of our previous work on query re-optimization and its results that motivate this research; Section IV presents the proposed query processing algorithm that uses machine learning for re-optimization decision; Section V discusses the experimental performance evaluation model and the results; and finally Section VI presents the conclusions and discusses future research directions.

II. Related Work

The problem of query re-optimization has been studied in the literature. In 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 [11, 4, 12, 2, 13]. These add additional overheads caused by query re-optimization to the overall performance of queries. Recently, learning techniques have been introduced to improve query optimization [14, 6, 15, 16, 17, 18, 19]. Feedback from the execution of past queries can be used to guide the query optimizer to produce better QEPs, so that the performance of future queries can be improved as a result. In this section, we review the work in the two related areas of adaptive query optimization and machine learning-based query optimization.

A. *Adaptive query optimization*

The idea of adaptive query optimization is that data statistics such as selectivity, cardinality, min and max values, and histograms are monitored during query execution. The execution can be paused whenever the algorithm decides to re-optimize the query. Then the new data statistics and intermediate results are used by the query optimizer to generate a new QEP and the query execution continues following the new QEP. In the works [11, 20], the authors manually set the checkpoints between certain operators of a QEP. These set points are placed based on rules defined by the authors, but whether or not a re-optimization call should be triggered is still based on cost estimation. Re-optimizations that take place at these check points are necessary but are still not accurate. The work in [1] proposes a query optimization method where the query is re-optimized multiple times during its execution based on stages. Every time one stage is finished, the query is re-optimized. We implemented this idea in our previous work and found that many re-optimizations are wasted [5], which we report in detail in Section III. Adaptive query optimization suffers from the following drawbacks: 1) It is hard to decide when a query should be re-optimized. Cost estimation can be a heuristic for the optimizer to make a decision but it is not accurate enough. Since re-optimization has a considerable overhead, re-optimizing a query during its execution when such re-optimization leads to no QEP changes can negatively impact the overall performance; and 2) Adjusting the QEP is difficult, especially on a cloud environment [21]. Some works such as [18] manually adjust the QEP, but these works suffer from a large search space for exploring the best adjustment for the current QEP.

B. Machine learning-based query optimization

Machine learning techniques have been used recently in query optimizations [22, 23, 7, 17, 6] for different purposes. In earlier works, Leo [23] 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. More recently, the work in [7] 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 [24], the authors examine the use of deep learning techniques in database research. Since then, reinforcement learning is also used. The work in [6] proposes a deep learning approach for cardinality estimation that is specifically designed to capture join-crossing correlations. SkinnerDB [22] and ReJoin [17] are two other works that use 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 whether a query re-optimization is beneficial in terms of query response time and monetary cost, which is the goal of our proposed algorithm.

III. Query Re-Optimization

To provide more details to support our motivations for the work proposed in this paper, in this section we report the findings we obtained when performing query re-optimization without using machine learning. In our previous work [5], 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 not only considers 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 to select the best plan. We presented this model in our previous work [25]. 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:

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

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 defined as follows:

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

where uw$_j$ and ew$_j$ describe the weight of the user and the environmental weight for objective j, respectively. 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. Fig. 2 shows the results of executing the query from our previous work [5]. In the experiment query, which is Query 1 shown in Fig. 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 of monetary cost is due to the fact that the more powerful containers that are selected to run the query are the containers which charge more hourly.

If the query is re-optimized only when such changes can be guaranteed, there will not be any unnecessary re-optimization. In order 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.

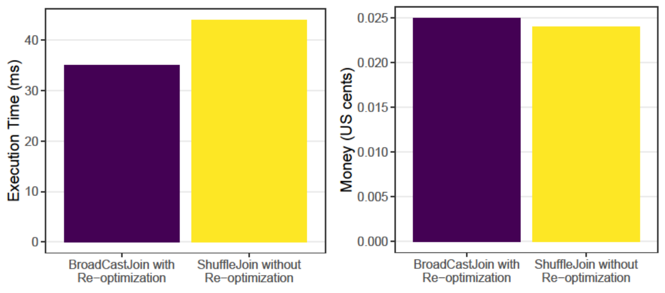

Fig. 2 Impacts of physical operators on time and monetary cost for execution of Query 1

IV. Machine Learning-Based Query Re-Optimization

In this section, we describe how machine learning is used in our query processing algorithm to predict when a query re-optimization is beneficial for the performance of queries. We first present an overview of our approach (Section IV.A), then we present its four parts: the feature selection (Section IV.B); the training data collection (Section IV.C); the machine learning model selection (Section IV.D); and the query processing algorithm that integrates with the machine learning-based re-optimization to optimize query response time and monetary cost (Section IV.E).

A. *Overview*

Fig. 3 shows the major steps of our proposed query processing 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 machine learning model is used here to take 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.

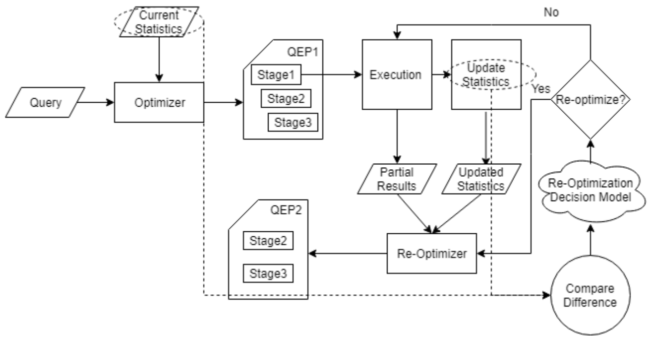

Fig. 3 Query processing of applying machine learning model

B. *Feature Selection*

In this section, we discuss what statistics (features) we collect to train our machine learning model. 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 be allowed to take place.

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.

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. 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.

These 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 $C_1, C_2, \ldots, C_n$ 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 as shown in Table I. 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, number of distinct values and the histogram influence the data size estimation [6, 12, 13]. 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.

C. *Training Data Collection*

First, we collect the training data by running random queries generated from all 22 types of queries in the TPC-H benchmark on our system and recording the data statistics, which are the values of the features we have selected in Section IV.B above. 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.

Fig. 4 shows the procedure of the training data collection. In order to better explain in detail how the training data is collected,

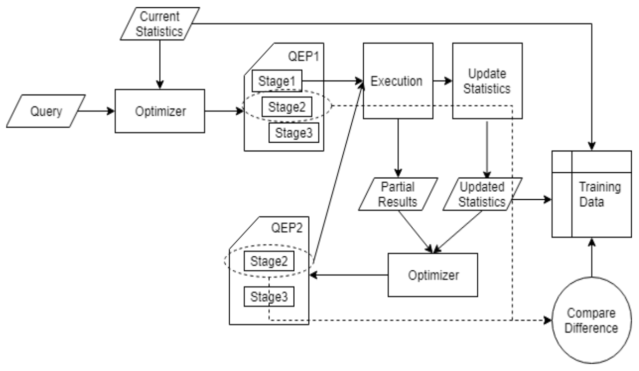

Fig. 4 The procedure of collecting the training data

we demonstrate an example of executing the following sample Query 2 shown in Fig. 5:

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

Fig. 5 Query 2

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. Fig. 6 shows the QEP generated by the query optimizer for Query 2. In Fig. 6, each node stands for different query operators. The arrows indicate the dataflow 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.

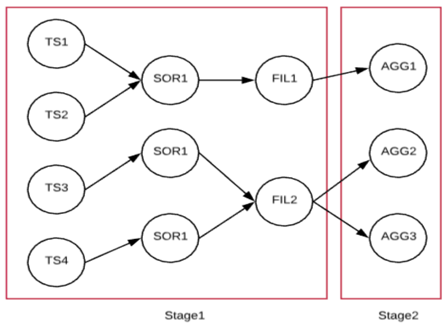

Fig. 6 QEP is divided into different stages after being compiled from the query

Then, Stage 1 is sent to the query execution engine. During the execution, we update the data statistics using the method mentioned in the work [1]. In this method, data statistics are collected during the execution and updated after the operators in one vertex finish. The vertex is similar to our stage. We call these updated statistics $Stat_{update}$. Since these statistics are collected from the actual running query, $Stat_{update}$ is more accurate than Statcurr which is obtained from the estimation. The difference between the $Stat_{update}$ and $Stat_{curr}$, is called $Stat_{diff}$. $Stat_{diff}$ includes the values of the features used as the training data. For example, the current selectivity and the updated selectivity of column A are 0.5 and 0.1, respectively, then the difference 0.4 is added as the value of the DIFF_SELECTIVITY feature in the training dataset. This process is applied to all the features. The selected features are shown in Table 1.

If the re-optimization is predicted to be beneficial, the QEP is then re-optimized using the updated data statistics. Following this, the next stage (Stage 2) is executed based on the new QEP. The process is then repeated for the rest of the stages. In this example, Stage 2 is possibly changed. At this point, Stage 2 after the re-optimization is compared to the Stage 2 before the re-optimization to observe any potential changes.

TABLE I. 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)

D. *Machine Learning Model Selection*

There exist a lot of machine learning models, but we need to choose a model that has a 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). 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 Decision Tree learning algorithm [8] may have a different prediction time overhead comparing with the prediction time overhead when applying a model trained by a Random Forest algorithm [26] 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 Decision Tree with 50 levels to derive a prediction is far different from checking a Decision Tree with 1000 levels.

In this paper, we study three major supervised machine learning algorithms which have also been used in the recent works on applying machine learning to database research [24]: Decision Tree, Random Forest, and Support Vector Machine (SVM). We compare them based on their prediction accuracy and prediction time and monetary overheads. The comparison results are reported in Section V.B and Section V.C.

E. *Query Processing Using the Proposed Machine Learning-based Re-optimization Model*

In this section, we illustrate how the trained model is applied during the query execution and the details are provided in Algorithm 1 shown in Fig. 7. In Lines 1 to 3 of Algorithm 1, the query is first optimized and converted to a QEP. This QEP is denoted as Old_QEP. At this time, the current data statistics are collected and stored as Old_Statistics. In Line 4, Stage 1 of the QEP is executed and removed from the QEP, and the intermediate results are saved. In Lines 5 and 6, the data statistics are updated and recorded as New_Statistics after the Stage 1's execution. In Line 7, the New_Statistics and Old_Statistics are compared and denoted as Diff_Statistics. From Line 8, while there are unfinished stages remaining in the Old_QEP, the Diff_Statistics is sent to the machine learning model to predict whether the new re-optimization will be beneficial or not and the corresponding decision of "YES" or "NO" is returned. If the decision is "YES", then in Lines 10 to 13, the Old_QEP is re-optimized into the New_QEP by the query optimizer using the new data statistics and the intermediate results. The New_QEP replaces the Old_QEP and the first stage in the New_QEP is then executed. If the decision is "NO" in Line 14, then the re-optimization is skipped and the current first stage of the Old_QEP is executed without re-optimization. From Line 16 to Line 22, regardless of whether the QEP has been re-optimized or not, the data statistics are always updated and compared as the model requires the Diff_Statistics to make a decision before executing the next stage. This procedure repeats until all the stages have been executed. Then the query results are sent to the user.

Algorithm: Query Processing with Machine Learning Based Re-Optimization (Re-Opt&ML)

```
INPUT: SQL query
OUTPUT: The query result set of the input query
1.  Old_Statistics = get current data statistics
2.  Result = Φ
3.  Old_QEP = generate_QEP (old_statistics, result)
4.  result = execute the first stage in Old_QEP and remove it
5.  Update the data statistics
6.  New_Statistics = get current data statistics
7.  Diff_Statistics = compute difference between Old_Statistics
    and New_Statistics
8.  while Old_QEP≠ Φ
9.      decision = RunPredictiveModel(Diff_Statistics)
10.     if decision = 'YES'
11.         New_QEP = generate_QEP(new_Statistics, result)
12.         result = execute the first stage in New_QEP and remove
    it.
13.         Old_QEP = New_QEP
14.     else if decision = 'NO'
15.         result = execute the first stage in Old_QEP and remove
    it
16.     end if
17.     if QEP ≠ Φ
18.         update data statistics
19.         Old_statistics = New_statistics
20.         New_statistics = get current data statistics
21.         Diff_ftatistics<-compute difference of Old_statistics and
    New_statistics
22.     end If
23. end while
24. return result
```

Fig. 7 Query processing algorithm with machine learning based re-optimization

V. Performance Evaluation

In this section, we first describe the hardware configuration and the procedure used to collect the training data. We then evaluate the machine learning models that predict whether the re-optimization is necessary to identify the best one to use in our query processing algorithm. Finally, we compare the performance of our query processing algorithm using the proposed machine learning model for re-optimization prediction with the performances of three competitive query processing algorithms: the optimal query processing algorithm, the query processing algorithm without using re-optimization, and the query processing algorithm using re-optimization after each stage is executed.

A. *Hardware Configuration*

There are two sets of machines that were 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 were used for the deployment of the query execution engine. Five 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 have 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 were modified from PostgreSQL 8.4. The data were distributed among these VPSs.

B. *Performance of different machine learning models*

1) Gathering training data: In order to study the model accuracy of different models on different sizes of training data, batches of different sizes of training data are created. Each batch contains a different number of queries that have been executed and monitored on the same system with the algorithm implemented based on our previous work [5]. The batch sizes are from 10,000 queries to 60,000 queries with an interval of 5,000 queries between them. To simulate the real usage of a database management system, the tuples of all tables are randomly changed constantly. This means we continuously insert, delete, or update tuples and index columns of all the tables. The table structures remain unchanged, no new tables are created, and no current tables are dropped. After each query execution, multiple observations are gathered as discussed in Section IV.C. After executing each query, multiple re-optimizations are conducted and each observation is for one of the query re-optimizations. Then these observations are labeled manually according to whether the QEP has been changed after the re-optimization. An observation is labelled "YES" if the

QEP has been changed after the re-optimization and “NO” otherwise.

2) *Tuning model parameters*: If the learning algorithm constructing the model has parameters, one important thing is to choose the best values for the parameters before training the model. For the Random Forest algorithm, we found three parameters influencing the model accuracy which are the number of trees, number of nodes in the tree, and random split option. In order to select the best values for the parameters for running this model, the Nested Cross Validation method *[27]* is used. This method first uses one round of cross validation to search for the best parameters’ values and then applies another round of cross validation to test the true accuracy of the model. In our experiment we found that using the number of trees as 600, the number of nodes in the tree as 200, and the random split option as YES provide the best accuracy of our Random Forest model.

3) *Model Accuracy:* Model accuracy reflects the overall success rate of predicting useful re-optimizations. We use 10-fold cross validation to test the accuracy of three models, Decision Tree, Random Forest, and SVM. Besides that, we study the impact of different data distribution on the accuracy of the learning models. We populate the data 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, 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 skew. We use skew data on purpose to see how model accuracy and query execution performance are impacted as shown in Fig. 8.

As we can see in Fig. 8, 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 between these three models are slightly different. Averagely, the Decision Tree 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 perform slightly different with the average accuracy being within 5% difference of each other.

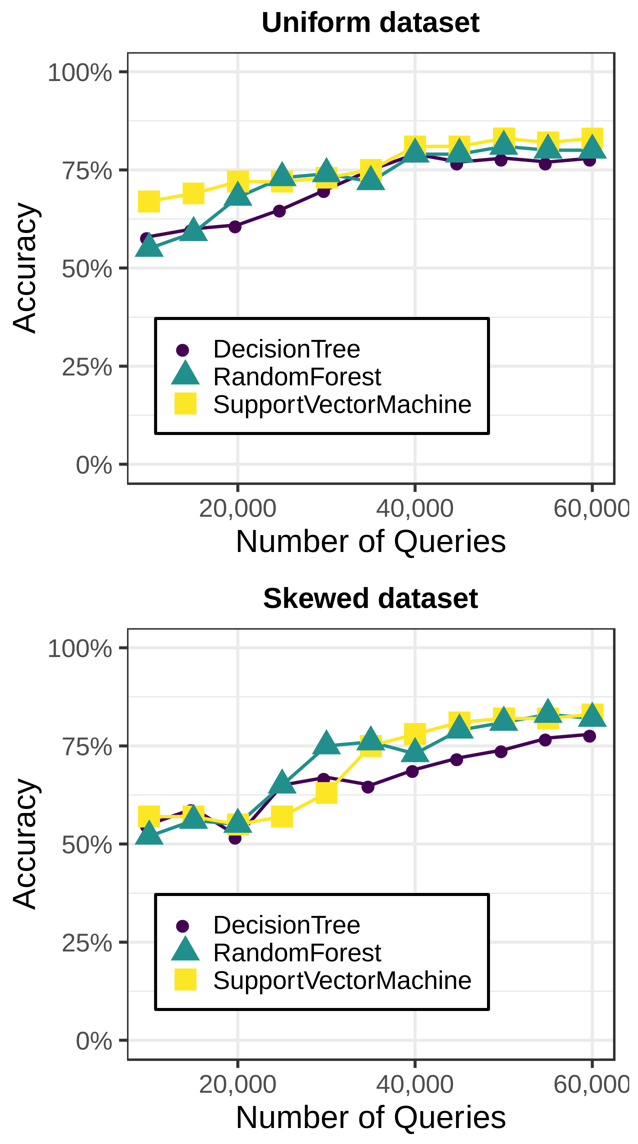

Fig. 8 Model accuracy of three different machine learning algorithms that learn from queries executed on (a) uniform data and (b) skewed data

C. *Performance of applying different machine learning models in query processing*

The model accuracy is close to each other as reported above; so to select which model should be used eventually, in this section, we evaluate these models in terms of performance on query execution when incorporating them into our query processing algorithm as shown in Algorithm 1 in Fig. 7 in Section IV.E. We generate 100 query instances from each of the 22 TPC-H benchmark query types, totaling 2200 queries. 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 cost when the query optimizer selects the best QEP. This means we assume the users have no preference on time or monetary cost themselves. The actual time and monetary cost resulted 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. Fig. 9 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 2.

From Table 2, we can see that SVM gives the best query response time. As shown in Fig. 8, 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 we have discussed in Section IV.D. As these models are applied online during query execution, the overhead caused by using these models is 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.

TABLE II. AVERAGE AND CUMULATIVE QUERY RESPONSE TIME AND MONETARY COST USING THREE DIFFERENT MACHINE LEARNING MODELS

	Decision Tree	*Random Forest*	*SVM*
Average Query Response Time	33 sec	35.4 sec	31.5 sec
Cumulative Query Response Time of 2,200 queries of 22 query types	72,600 sec	77,880 sec	69,300 sec
Average Query Monetary Cost	0.068 ¢	0.071 ¢	0.069 ¢
Cumulative Query Monetary Cost of 2,200 queries of 22 query types	149.6¢	156.2 ¢	151.8 ¢

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 Fig. 9 (b). 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 of money more 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 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 query processing algorithm and compare this algorithm against other query processing algorithms. We select this model for comparison purposes only; we do not intent to suggest which model should be selected automatically as some QEPs may be

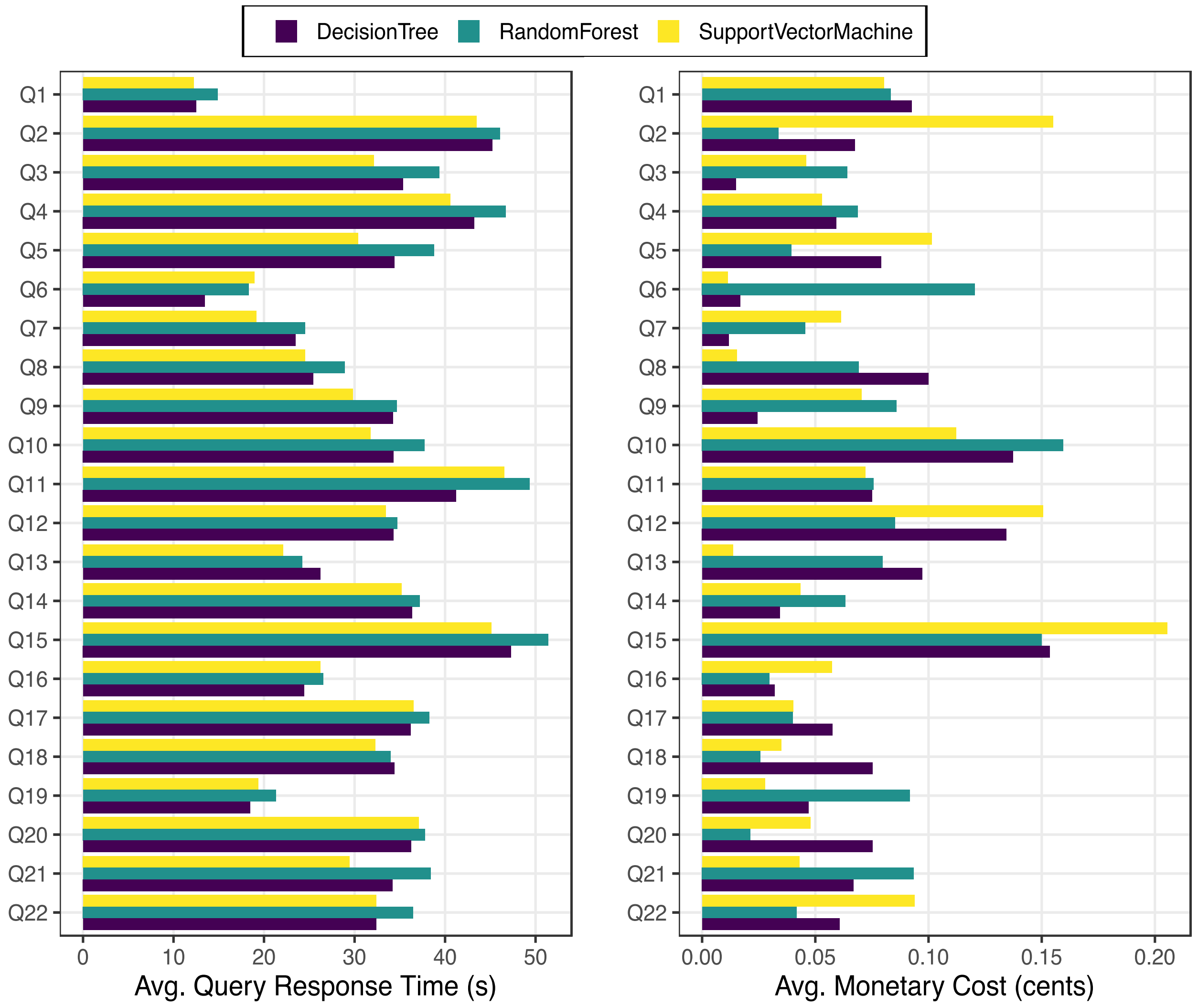

Fig. 9 (a) and (b). Average response time and Average monetary cost of executing queries using three different machine learning models executed faster but costs more money and vice versa, depending on the selected model.

D. Performance of different query processing algorithms

In this section, we compare our proposed query processing algorithm that uses the proposed machine learning-based model for re-optimization (denoted as Re-Opt&ML) with three algorithms: 1) optimal query processing (denoted as Optimal); 2) query processing that does not use re-optimization (denoted as NoRe-Opt); and 3) query processing that uses re-optimization after each stage is executed (denoted as Re-Opt). For the Optimal, we manually change the cost estimations used by the query optimizer to the actual query execution cost so that the optimizer can always get the most accurate cost estimation, which is not possible in practice but provides a baseline of comparison. For the NoRe-Opt algorithm, we execute queries with no re-optimization at all, which is the way that many current commercial database systems do. For the Re-Opt algorithm, we re-optimize every query automatically after the completion of each stage in the query.

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 averagely large differences between Re-Opt and Re-Opt&ML so that we can see with the help of machine learning, how much improvement can be obtained with re-optimization.

1) Skew Data: We first compare each of the three algorithms with Optimal. From Fig. 10, where queries are executed on skew data, NoRe-Opt, Re-Opt, and Re-Opt&ML give 2.6x, 2x, and 1.7x longer query response time than Optimal, and NoRe-Opt, Re-Opt, and Re-Opt&ML spend 1.1x, 1.4x, and 0x more money than Optimal, respectively. Comparing our algorithm with NoRe-Opt and Re-Opt on skew data, from Fig. 10 (a) we see that Re-Opt&ML yields 13% less query response time than Re-Opt and 35% less query response time than NoRe-Opt, while Re-Opt gives 23% less query response time than NoRe-Opt. In terms of monetary cost, from Fig. 10 (b) Re-Opt&ML spends 34% less money than Re-Opt and 17% less money than NoRe-Opt, while Re-Opt spends 23% more money than NoRe-Opt.

The above results show that Re-Opt&ML can save more time and monetary cost than the other two algorithms, NoRe-Opt and Re-Opt, and Re-Opt spends less time than NoRe-Opt, while spends more money than NoRe-Opt. These differences apply to the different query types as shown in Fig 11. These query types contain a lot of JOIN operators or subqueries. 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. Also, re-optimizations help to decide the degree of parallelism of each operator so that unnecessary operators can be avoided which saves a lot of money as fewer containers are used for executing these operators. Also, using machine learning further helps avoid unnecessary re-optimizations.

2) Uniform Data: In addition to the results obtained on executing queries on skew data, Fig. 10 (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 the large differences in query response time and monetary cost. Similar to the case of skew data, here we also first compare each algorithm with Optimal. From Fig. 10 (c) we see that NoRe-Opt, Re-Opt, and Re-Opt&ML give 2.6x, 2.5x, and 2x more query response time than Optimal, respectively. The query response times resulted from Re-Opt&ML and Re-Opt on uniform data are more 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 on executing queries on skew data.

In terms of monetary cost, we see that NoRe-Opt, Re-Opt, and Re-Opt&ML spend 1.2x, 4.3x, and 1.3x more money than Optimal. The monetary cost incurred by Re-Opt is noticeably higher than that incurred by the other algorithm. As re-optimization is not helpful on executing queries on uniform data, a lot of unnecessary re-optimizations waste a large amount of time.

Comparing our algorithm with NoRe-Opt and Re-Opt, from Fig. 10 (c) we see that Re-Opt&ML yields 14% less query response time than Re-Opt and yields the same time as NoRe-Opt, while Re-Opt gives 7% more time than NoRe-Opt. In term of monetary cost, from Fig. 10 (d) we see Re-Opt&ML spends the same amount of money as Re-Opt and 7% more money than NoRe-Opt, while Re-Opt spends 10% more money than NoRe-Opt. From these results, we find that when queries are executed on uniform data, re-optimization does not perform as well as on skew data. The monetary cost is even higher for re-optimization algorithms; but still, Re-Opt&ML saves more time compared to Re-Opt.

In summary, we conclude that using machine learning to predict when a re-optimization is beneficial does improve query response time no matter queries are executed on 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.

VI. Conclusion And Future Work

This paper presents a query processing algorithm that uses a machine learning-based model to decide whether or not a query should be re-optimized. The experiments conducted show that for skew data, this algorithm improves the performance of queries in terms of both query response time and monetary cost and outperforms the existing algorithms that use either no re-optimization or re-optimization after each stage in the query execution plan (QEP) is executed. For uniform data, this algorithm yields better query response time than the one that uses re-optimization without machine learning, but does not improve monetary cost.

While our studies have shown that machine learning has a positive impact on deciding whether a re-optimization should be conducted, the 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 (features) which may not be available in all

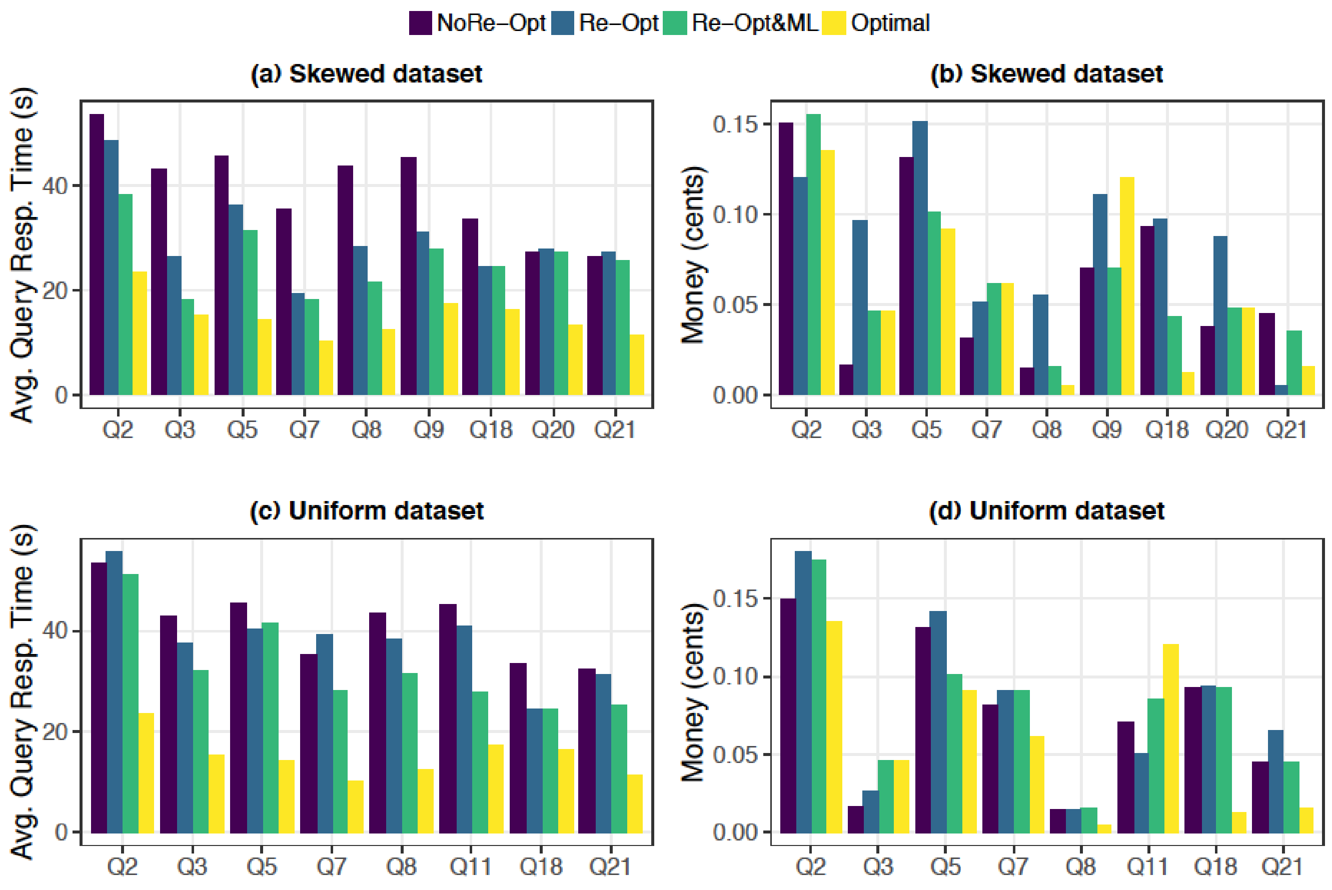

Fig. 10 (a)-(d). Average query response time and monetary cost of executing one query from different query types on skew data (a) and (b) and on uniform data (c) and (d)

DBMSs. For future work, we will investigate techniques that do not rely on data statistics. In addition, we will also extend our approach to predicting, independently of query stages, when a query re-optimization should be carried out, and predicting how many times such query re-optimization should occur.

Acknowledgement

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

References

[1] N. Bruno, S. Jain and J. Zhou, "Continuous cloud-scale query optimization and processing," *VLDB*, 2013.

[2] F. Wolf, N. May, R. Willems and K.-U. Sattler, "On the Calculation of Optimality Ranges for Relational Query Execution Plans," in *SIGMOD*, 2018.

[3] A. Deshpande, Z. Ives and V. Raman, "Adaptive Query Processing," in *VLDB 1,1*, 2007.

[4] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh and M. Cilimdzic, "Robust query processing through progressive optimization.," in *SIGMOD*, 2004.

[5] C. Wang, Z. Arrani, L. Gruenwald and d. Laurent, "Adaptive Time,- Monetary Cost Aware Query Optimization on Cloud DataBase," 7th Scalable Cloud Data Management, 2018.

[6] 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.

[7] W. Chenggang, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao and S. Rao, "Towards a learning optimizer for shared clouds," in *VLDB Endow. 12, 3*, November,2018.

[8] J. Quinlan, "Induction of decision trees," in *Machine Learning 1,(1)*, March,1986.

[9] C. Corinna and V. Vladimir N., "Support-vector networks," in *Machine Learning. 20 (3)*, 1995.

[10] 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*, July, 2019.

[11] Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. S. Weld, "An adaptive query execution system for data integration," in *SIGMOD '99*, 1999.

[12] W. Wu, J. F. Naughton and H. Singh, "Sampling-Based Query Re-Optimization," in *SIGMOD*, 2016.

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

[14] A. A. Bankole and S. A. Ajila, "Predicting cloud resource provisioning using machine learning techniques," in *Canadian Conference on Electrical and Computer Engineering*, 2013.

[15] A. Jindal, K. Karanasos, S. Rao and H. Patel, "Thou Shall Not Recompute: Selecting Subexpressions to Materialize at Datacenter Scale," in *VLDB Endow. 11, 7*, March,2018.

[16] H. Liu, M. Xu, Z. Yu, V. Corvinelli and C. Zuzarte, "Cardinality Estimation Using Neural Networks," in *25th Annual International Conference on Computer Science and Software Engineering*, 2015.

[17] R. Marcus and O. Papaemmanouil, "Deep Reinforcement Learning for Join Order Enumeration," in *1st International Workshop on Exploiting Artificial Intelligence Techniques for Data Management,*, 2018.

[18] P. Yongjoo, T. Ahmad, C. Michael and M. Barzan, "Database Learning: Toward a Database that Becomes Smarter Every Time," in *SIGMOD*, 2017.

[19] J. Ortiz, M. Balazinska, J. Gehrke and S. S. Keerthi, "Learning State Representations for Query Optimization with Deep Reinforcement Learning," in *SIGMOD*, 2018.

[20] K. Navin and D. David, "Efficient mid-query re-optimization of sub-optimal query execution plans," in *SIGMOD*, 1998.

[21] W. Lang, R. Nehme and I. Rae, "Database optimization for the cloud: Where costs, partial results, and consumer choice meet," in *CIDR*, 2015.

[22] I. Trummer, S. Moseley, D. Maram, S. Jo and J. Antonakakis, "SkinnerDB: Regret-bounded Query Evaluation via Reinforcement Learning," in *PVLDB, 11(12)*, 2018.

[23] M. Stillger, G. M. Lohman, V. Markl and M. Kandil, "LEO - DB2's LEarning Optimizer," in *VLDB '01*, 2001.

[24] W. Wang, M. Zhang, G. Chen, H. V. Jagadish, B. C. Ooi and K.-L. Tan, "Database Meets Deep Learning: Challenges and Opportunities," in *SIGMOD Rec., 45(2)*, September, 2016.

[25] 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.

[26] L. Breiman, "Random Forests," in *Machine Learning, 45 (5)*, 2001.

[27] E. Alpaydin, in *Introduction to Machine Learning 3rd Edition*, 2014.