Budget-aware Query Tuning: An AutoML Perspective

Wentao Wu Chi Wang
Microsoft Research
{wentao.wu, wang.chi}@microsoft.com

ABSTRACT

Modern database systems rely on cost-based query optimizers to come up with good execution plans for input queries. Such query optimizers rely on *cost models* to estimate the costs of candidate query execution plans. A cost model represents a function from a set of *cost units* to query execution cost, where each cost unit specifies the *unit cost* of executing a certain type of query processing operation (such as table scan or join). These cost units are traditionally viewed as *constants*, whose values only depend on the platform configuration where the database system runs on top of but are invariant for queries processed by the database system. In this paper, we challenge this classic view by thinking of these cost units as *variables* instead. We show that, by varying the cost-unit values one can obtain query plans that significantly outperform the default query plans returned by the query optimizer when viewing the cost units as constants. We term this cost-unit tuning process "query tuning" (QT) and show that it is similar to the well-known hyper-parameter optimization (HPO) problem in AutoML. As a result, any state-of-the-art HPO technologies can be applied to QT. We study the QT problem in the context of *anytime tuning*, which is desirable in practice by constraining the total time spent on QT within a given budget—we call this problem *budget-aware* query tuning. We further extend our study from tuning a single query to tuning a workload with multiple queries, and we call this generalized problem budget-aware workload tuning (WT), which aims for minimizing the execution time of the entire workload. WT is more challenging as one needs to further prioritize individual query tuning within the given time budget. We propose solutions to both QT and WT and experimental evaluation using both benchmark and real workloads demonstrates the efficacy of our proposed solutions.

1. INTRODUCTION

Modern database systems rely on cost-based query optimizers to come up with good execution plans for input queries. Such query optimizers rely on *cost models* to estimate costs of candidate query execution plans. A cost model represents a function from a set of *cost units* to the query execution cost, where each cost unit specifies the *unit cost* of executing a certain type of query processing operation (such as table scan or join). For instance, Table 1 presents some of the cost units used by PostgreSQL's query optimizer [1]. This paradigm is followed by other mainstream query processing engines as well, such as Microsoft SQL Server [8], IBM DB2 [25], and Apache Spark [15]. These cost units are traditionally viewed as *constants* [1], whose values only depend on the *platform configuration* (e.g., CPU speed) where the query processing engine runs on top of and need to be *calibrated* against the platform [11,26,34,35,36,38]; however, they are *invariant* regardless of the queries being processed by the database system.

Cost Unit	Default Value
seq_page_cost	1.0
random_page_cost	4.0
cpu_tuple_cost	0.01
cpu_index_tuple_cost	0.005
cpu_operator_cost	0.0025
parallel_tuple_cost	0.1

Table 1: Examples of cost units used by PostgreSQL's query planner/optimizer [1].

In this paper, we challenge this classic view by thinking of these cost units as *variables* instead that can be changed across queries. We show that, by varying the cost-unit values one can obtain query plans that significantly outperform the default query plans returned by the query optimizer when viewing the cost units as constants. We call this per-query cost-unit tuning process "query tuning" (QT). At a high-level, query tuning is similar to the *hyper-parameter optimization* (HPO) problem [39] from *automated machine learning* (a.k.a. AutoML), which has received tremendous attention in recent years [13]. While the HPO probem aims to find appropriate hyper-parameter values, such as the learning rate and batch size when using stochastic gradient descent (SGD) to train a deep neural network, that can improve the quality of the ML model, QT shares the similar goal of improving the quality of the query plan by seeking appropriate values of the cost units.

The similarity between QT and HPO implies that existing HPO technologies can be directly applied to QT. However, a straightforward application is less attractive, as most of the existing HPO technologies are not *budget-aware*, namely, they do not constrain themselves to conform to a user-specified *tuning time budget*. There are so far only a few exceptions, including Hyperband [20], BOHB [12], CFO [33], and BlendSearch [30], which allow user to specify a *timeout* and will exit HPO once the timeout is reached. However, in practice one often wants to tune multiple queries (called a workload) altogether instead of tuning just one single query. None of the above work on HPO can be directly applied to this multi-query workload tuning (WT) problem.

In this paper, we study the budget-aware QT and WT problems. We propose solutions to both problems and evaluate their efficacy using both benchmark and real workloads. The query plans found by our current solutions can significantly improve over the default query plans generated by the query optimizer while conforming to the tuning time budget.

The idea of tuning query performance has been extensively explored in the literature but in different sense compared to the one proposed in this paper. Lots of recent work has been devoted to the so-called *knob tuning* (see [40] for a recent survey). Some existing work, e.g., UDO [31], also views the cost units studied in this paper as tunable knobs, though we think the cost units should be separated from other knobs and are worth its own treatment for the following reasons. First, some of the knobs are related to the runtime configurations of the database server, e.g., buffer pool size. Changing those knobs often requires a server restart that may not be feasible in many situations (e.g., cloud database services with stringent SLA's). In contrast, tuning the cost units does not require a server restart and therefore does not pose significant impact on the runtime state of the database server. Second, while there are also other knobs that do not require server restart, e.g., *max_parallel_workers* for PostgreSQL, such knobs only affect the running time of the query plan that has already been chosen by the query optimizer. In contrast, cost units can affect the decision made by the query optimizer in terms of choosing which query plan for execution. This is indeed a more fundamental distinction when viewing cost units as tunable knobs. In this spirit, the cost units can be thought of as a certain type of query hint [7]. However, the goal of query hint is to constrain the search space of the query optimizer to avoid bad query plans that could have been proposed by the optimizer without such restrictions; on the other hand, tuning cost units actually gives the optimizer more freedom in terms of proposing query plans, and the decision of picking the best plan is deferred to the moment when actual plan execution time is observed. Due to their overheads, the query/workload tuning technologies studied in this paper can perhaps only be applied to *recurring* queries/workloads, where one can tune the queries/workloads in an offline manner and pay it as a one-time price [14]. Nonetheless, given the strong connection between QT and HPO, we believe that more progress can be made in the future to enable QT for tuning more adhoc workloads in an online fashion.

2. PROBLEM FORMULATION

We assume that a query optimizer uses a cost model configured with a set of tunable parameters, referred to as *cost units* (ref. Table 1 for examples), to estimate the cost of a candidate query execution plan. The plan with the lowest estimated cost is returned by the optimizer for final query execution. Without loss of generality, we use $\vec{u}$ to represent the set of tunable cost units as an ordered vector. Given a query q, we use $P(q, \vec{u})$ to represent the query plan returned by the query optimizer with the cost units $\vec{u}$. Different values of $\vec{u}$ may therefore yield different query plans. We next formulate the problems of budget-aware query tuning and workload tuning.

2.1 Budget-aware Query Tuning

Let q be a specific query, and let U be the search space of $\vec{u}$. That is, if $\vec{u} = (u_1, ..., u_m)$ where each u_i is within some range/domain D_i (e.g., $u_i \in [a_i, b_i]$), then $U = \prod_{i=1}^{m} D_i$. This covers both discrete and continuous cost units, though in practice cost units are typically continuous (within certain ranges).

Let B be a given budget on the tuning time. Let $\vec{u}_1$, ..., $\vec{u}_K$ be the successive trials on the cost units. Let $t(q, \vec{u}_j)$ be the execution time of the corresponding query plan $P(q, \vec{u}_j)$, for $1 \leq j \leq K$. The *budget constraint* can then be expressed as $\sum_{j=1}^{K} t(q, \vec{u}_j) \leq B$. The problem of budget-aware query tuning is defined as:

Definition 1 (Budget-aware Query Tuning)**.** *Find* $\vec{u}^* = \text{argmin}_{1 \leq j \leq K}\{t(q, \vec{u}_j)\}$ *w.r.t.* $\sum_{j=1}^{K} t(q, \vec{u}_j) \leq B$.

2.2 Budget-aware Workload Tuning

Let $W = \{q_1, ..., q_n\}$ be a workload of n queries. Let f_i be the *frequency* of the query q_i being tuned. For q_i, we define its total tuning time $t_i = \sum_{j=1}^{f_i} t(q_i, \vec{u}_{ij})$, where $\vec{u}_{ij}$ represents (the cost units of) the j-th trial of q_i. The budget constraint at workload-level can then be expressed as $\sum_{i=1}^{n} t_i \leq B$.

Definition 2 (Budget-aware Workload Tuning)**.** *Find*

$$\vec{u}_i^* = \text{argmin}_{1 \leq j \leq f_i}\{t(q_i, \vec{u}_{ij})\}$$

for $1 \leq i \leq n$ *w.r.t. the budget constraint* $\sum_{i=1}^{n} t_i \leq B$.

Remark. The above definition automatically minimizes the workload execution time w.r.t. the budget constraint

on tuning time, which can be expressed as

$$t(W, \vec{u}_W^*) = t(q_1, ..., q_n, \vec{u}_1^*, ..., \vec{u}_n^*) = \sum_{i=1}^{n} t(q_i, \vec{u}_i^*).$$

Discussion. An alternative of Definition 2 is to find one set of cost units for the entire workload, instead of one set of cost units for each individual query. Specifically, let $\vec{u}_1$, ..., $\vec{u}_K$ be the successive trials on the cost units for the entire workload W, and let $t(W, \vec{u}_j) = \sum_{i=1}^{n} t(q, \vec{u}_j)$ be the total workload execution time.

Definition 3 (Query-independent Budget-aware Workload Tuning)**.** *Find* $\vec{u}^* = \text{argmin}_{1 \le j \le K}\{t(W, \vec{u}_j)\}$ *w.r.t. the budget constraint* $\sum_{j=1}^{K} t(W, \vec{u}_j) \le B$.

When the budget B is sufficient, Definition 2 is more general than Definition 3, because we should be able to get the same or a better query plan for each query under Definition 2. To see this, notice that the best sets of cost units for individual queries are not necessarily the same, which, on the other hand, is an implicit constraint under Definition 3. It remains interesting to empirically compare Definitions 2 and 3 when budget is limited, and we leave this as one direction for future work.

3. PROPOSED SOLUTIONS

Below we propose solutions to budget-aware query tuning (QT) and workload tuning (WT).

3.1 Budget-aware Query Tuning

Since QT is similar to HPO, in theory any HPO algorithm $\mathcal{A}$ can be adapted to work for QT. For example, *random search* is a simple but competitive algorithm for HPO [5]. It can be easily customized to a budget-aware algorithm by monitoring the time spent on each random trial and terminating once the timeout is reached.

Optimization by Plan Caching. Since different cost units may result in the same query execution plan, it is possible that some plan $P(q, \vec{u}_j)$ is a duplicate of another plan $P(q, \vec{u}_i)$ $(i < j)$ in the sequence $\vec{u}_1$, ..., $\vec{u}_K$. One optimization is therefore to maintain a *cache* for observed plans, if memory is not constrained. The tuning time of a duplicate plan is set to zero.

Optimization by Early Stopping. Since only the optimal $\vec{u}^*$ matters, we can safely stop executing a plan $P(q, \vec{u}_j)$ if $t(q, \vec{u}_j) \ge t(q, \vec{u}_j^*)$, where $t(q, \vec{u}_j^*)$ is the lowest execution time observed up to the j-th trial. In practice, we usually have a default value $\vec{u}_0$ for $\vec{u}$ (e.g., the built-in values such as the ones shown in Table 1 for PostgreSQL). As a special case of the above "early stopping" idea, we can stop executing $P(q, \vec{u}_j)$ if $t(q, \vec{u}_j) \ge t(q, \vec{u}_0)$. This is also a worst-case scenario, as we have $t(q, \vec{u}_j^*) \le t(q, \vec{u}_0)$, obviously.

3.2 Budget-aware Workload Tuning

WT is more complicated than QT, as one has to decide which query to tune next while conforming to the total budget on tuning time. We propose the following four strategies: (1) round robin; (2) cost-based prioritization; (3) multi-armed bandit; (4) improvement rate. Moreover, both the plan-cache based optimization and the early-stopping optimization can be used for WT.

3.2.1 Round Robin

The round robin strategy simply rotates among the queries and stops when the tuning time budget is exhausted. Albeit a simple strategy, it has been deemed as a robust and strong baseline in the literature [21].

3.2.2 Cost-based Prioritization

This strategy is inspired by the idea of using a priority queue to prioritize query tuning in Microsoft's Database Tuning Advisor (DTA) [9]. Specifically, we order the queries by their best execution time observed so far and then select the slowest query to tune next.

3.2.3 Multi-armed Bandit

This strategy models workload tuning as a multi-armed bandit problem. Specifically, we view each query as an arm, and use the well-known UCB1 score [2, 3] as the criterion for selecting the next query to tune:

$$\underset{q}{\text{argmax}} \left[\bar{r}(q) + \lambda \cdot \sqrt{\frac{\ln N}{f_q}} \right].$$

Here, λ is a constant that balances *exploration* and *exploitation*. We choose $\lambda = \sqrt{2}$ as suggested in the literature [17]. f_q is the number of times (i.e. frequency) that query q is tuned, and $N = \sum_{q \in W} f_q$. $\bar{r}(q)$ is the *average reward* of q. The reward $r(q, \vec{u})$ of tuning q with cost units $\vec{u}$ is defined as its relative improvement over the query execution time with the default cost units $\vec{u}_0$: $r(q, \vec{u}) = \max\{1 - \frac{t(q,\vec{u})}{t(q,\vec{u}_0)}, 0\}$. That is, we cap the reward at 0 if $\vec{u}$ is even worse than $\vec{u}_0$. This should not occur if the early-stopping optimization is used. The average reward is therefore $\bar{r}(q) = \frac{1}{f_q} \sum_{j=1}^{f_q} r(q, \vec{u}_j)$. Again, we stop when the tuning time budget is exhausted.

3.2.4 Improvement Rate

The improvement $I_j(q)$ of a query q is defined as the gap between its best execution time found so far and the default execution time with cost units $\vec{u}_0$. That is, $I_j(q) = \max\{t(q, \vec{u}_0) - t(q, \vec{u}_j^*), 0\}$, for $1 \le j \le f_q$. Again, we cap the improvement at 0 if $\vec{u}_j^*$ is worse than $\vec{u}_0$, which should not happen if the early-stopping optimization is used. The improvement rate is then defined as $R_j(q) = \frac{I_j(q)}{f_q}$. This strategy always selects the query with the highest improvement rate to tune next, which is inspired by BlendSearch [30]. Again, we stop when the tuning time budget is exhausted.

3.2.5 Summary and Discussion

The strategies discussed above are by no means perfect or exhaustive. We briefly discuss potential improve-

Name	DB Size	Queries	Tables	Joins	Scans
JOB	9.2GB	33	21	7.9	8.9
TPC-DS	sf=10	99	24	7.7	8.8
Real-A	100GB	25	20	6.5	7.2
Real-B	60GB	16	7	1.9	2.9

Table 2: Database and workload statistics.

ments, extensions, and other alternatives. First, it is not necessary to always start running these strategies from scratch. As we mentioned, there has been prior work on calibrating cost units (e.g., [11,26,35]) by viewing them as constants. These refined cost constants can be used as starting points of the above strategies for further fine-tuning. Second, the cost-unit calibration technologies could themselves serve as solutions to the WT problem, especially for its query-independent variant (see Definition 3), though in a budget-unaware sense. However, these technologies typically come with additional implementation overhead as well as subtleties that the WT strategies proposed in this paper do not have. For example, one technique used in [35] is to design a set of *independent* "calibration queries" that can target individual cost units. This is relatively easy for some database systems such as PostgreSQL but becomes more challenging for others such as Microsoft SQL Server. Not only does Microsoft SQL Server have many more cost units compared to PostgreSQL, but some of its cost units are also *correlated*. Since no calibration query can separate two correlated cost units, the technique from [35] needs to be extended, which requires further research.

4. EXPERIMENTAL EVALUATION

We report experimental results on evaluating the performance of our proposed budget-aware query and workload tuning technologies.

Datasets and Workloads. We used various benchmark and real workloads in our evaluation. For benchmark workloads, we use the join order benchmark (**JOB**) [18], as well as the **TPC-DS** benchmark with scaling factor 10. **JOB** contains 113 query instances in total, which are grouped into 33 templates, and we pick one query instance from each template. We use the same protocol for **TPC-DS**. We choose these two benchmark workloads due to the diversity and complexity in their queries that offer more opportunities for finding better query plans via query tuning. We also use two real workloads, denoted by **Real-A** and **Real-B**. Table 2 summarizes the key statistics of these workloads. The last two columns represent the average number of joins and table scans contained by a query in the workload.

Experimental Settings. We perform all experiments using Microsoft SQL Server 2017 under Windows Server 2022, running on a workstation equipped with 2.3 GHz AMD CPUs and 256 GB main memory. We focus on tuning eight cost units that are critical to the costs of workhorse operators such as *table scan*, *index seek*, *sort*, *hash join*, and *nested-loop join*. For each cost unit c, we set its range/domain as $[0.1 \times c_d, 10 \times c_d]$ for exploration (ref. Section 2.1), where c_d is the default value of c.

Name	Default Plan (minutes)	Best Plan (minutes)	Percentage Improvement (%)	Tuning Time (minutes)
JOB	3.72	1.33	64.2%	130
TPC-DS	4.74	3.24	31.6%	580
Real-A	13.96	7.36	47.3%	675
Real-B	13.21	8.38	36.5%	160

Table 3: Summary of query tuning results

4.1 Query Tuning Results

We use *percentage improvement* as the performance metric, defined as $1 - \frac{t(P_{\text{best}})}{t(P_{\text{default}})}$. $t(P_{\text{best}})$ and $t(P_{\text{default}})$ represent the execution time of the best plan found by QT and that of the default plan chosen by the query optimizer (using the built-in values of the cost units).

For each query, we give the HPO algorithm $\mathcal{A}$ 100 trials, and report the best plan found. For the HPO algorithm $\mathcal{A}$, we evaluated both *random search* [5] and *SMAC* [16], but we found that their performances were similar. As a result, we only report results by using random search, due to its simplicity and lower overhead.

Table 3 summarizes the QT results for the workloads evaluated. We observe percentage improvement ranging from 31.6% to 64.2% at workload level (i.e., the total execution time of all queries). Figure 1 further showcases the percentage improvement for each query in the **JOB** workload, whereas Figure 2 presents the corresponding execution time (in seconds) of the default plan and the best plan found by QT. Note that we did not test the budget-aware version of QT, as it is a special case of budget-aware WT that will be covered in Section 4.2.

Case Study. We further present a case study of the **JOB** query #5, which shows significant improvement in Figure 2. We observe that the most significant parameter changes when tuning this query happen in cost units related to random reads and index seeks: the values of these cost units are significantly decreased (by 5×) for the best plan found by QT. Since the **JOB** database size (ref. Table 2) is much smaller than the server's memory size (i.e., 256 GB), most of the database can be cached in the main memory. As a result, random reads and index seeks are much faster compared to the case when the database resides on disk. This sheds some light on the significant improvement observed for this query.

Discussion. From Table 3, the improvements across workloads vary. Clearly, the improvement is determined by the gap between $t(P_{\text{default}})$ and $t(P_{\text{best}})$. One important factor that contributes to this gap is query complexity: for a simple query with no joins, the gap is likely small; for a complex query with many joins, the gap is likely large. Meanwhile, the degree of cardinality estimation (CE) errors also matters, as it is well-known

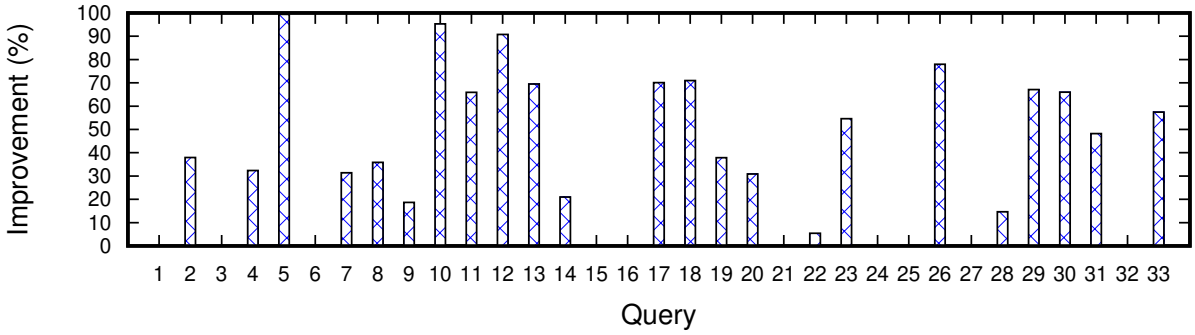


Figure 1: Percentage improvement of each query in the JOB workload.

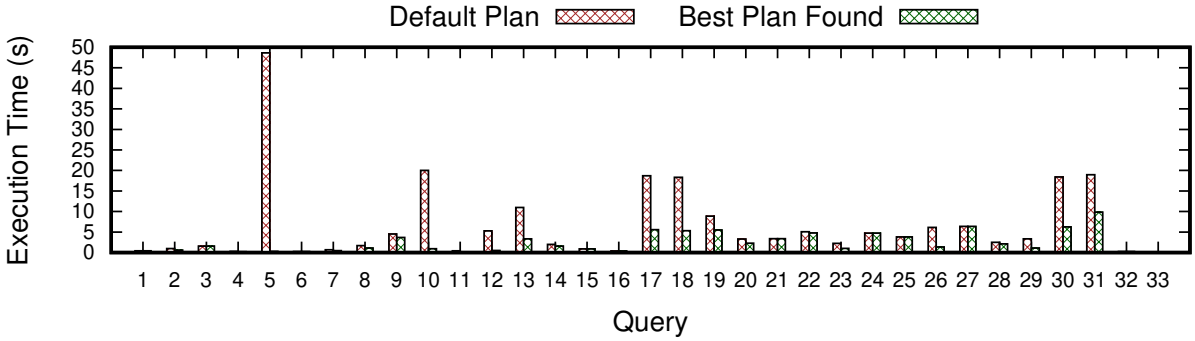


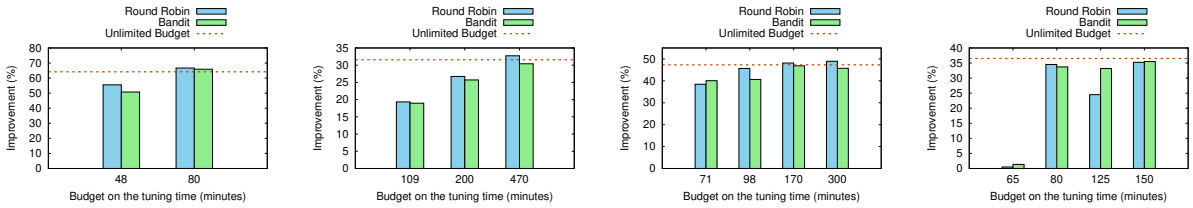
Figure 2: Execution time of default plan vs. best plan found by QT for each query of the JOB workload.

that query optimizers are likely to choose poor execution plans in the presence of CE errors [24]. Hence, we would expect larger improvements for workloads with complex queries and significant CE errors. From the results reported in Table 3, we observe the largest percentage improvement on the **JOB** workload. This does not seem like a coincident, as **JOB** is intentionally designed to challenge the cardinality estimators of query optimizers with complex join queries (ref. Table 2) [19].

4.2 Workload Tuning Results

We vary the budget on the time given to the workload tuning algorithms and test the percentage improvement at workload-level. Figure 3 presents the results on the workloads tested. The x-axis represents the tuning time given to the algorithms, whereas the y-axis reports the percentage improvement observed. To avoid clutter, Figure 3 only includes the two best-performing algorithms, *round robin* and *multi-armed bandit*, which significantly outperform the other two algorithms, *cost-based prioritization* and *improvement rate* (see Figure 4 for a comparison on **TPC-DS**). Moreover, the dashed line in each chart represents the percentage improvement observed in the QT experiment with 100 trials of random search for each query, whereas the corresponding tuning time has been reported in the last column of Table 3. We observe that we can obtain similar percentage improvement with much less tuning time. For example, on **JOB** it took only 80 minutes for both algorithms to achieve the same percentage improvement, compared to the 130 minutes in QT (i.e., 38.5% reduction); on **Real-A**, it took only 98 minutes for *round robin* to achieve the same percentage improvement, in contrast to the 675 minutes taken in QT (i.e., 85.5% reduction).

Comparison of Workload Tuning Algorithms. Figure 4 compares the four workload tuning algorithms with varying tuning time budget, using the **TPC-DS** workload. We observe that *round robin* and *multi-armed bandit* perform similarly, and they significantly outperform the other two algorithms, *cost-based prioritization* and *improvement rate*. The *cost-based prioritization* strategy often does not perform well in a budget-aware setting, because it can often get stuck on some very expensive query that is also hard to improve. The *improvement rate* strategy bypasses this issue by focusing on tuning queries that can improve quickly. As a result, it improves over the *cost-based prioritization* strategy. However, it remains less effective compared to *round robin* and *multi-armed bandit*. On the other hand, it is somewhat surprising to see *round-robin* performs closely to the best-performing *multi-armed bandit* strategy in most of the cases. One reason could be that, for the workloads that we tested, all queries are worth tuning and there is little need to prioritize. However, we do not expect that this property holds in general, and we expect that the *multi-armed bandit* strategy should outperform *round-robin* for workloads with more heterogeneous queries. The question of designing a better strategy for budget-aware workload tuning (other than the ones studied in this paper, which are somewhat standard) remains open and we intend to leave it for future work.



(a) **JOB** (b) **TPC-DS** (c) **Real-A** (d) **Real-B**

Figure 3: Percentage improvement resulted from workload tuning when varying the budget on tuning time.

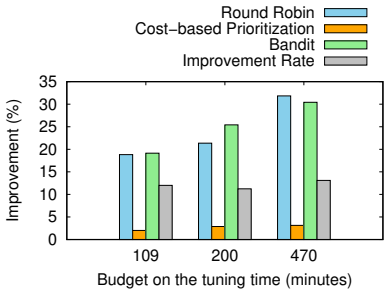


Figure 4: Comparison of WT algorithms on TPC-DS

5. RELATED WORK

Autonomous Knob Tuning. The problem of knob tuning for database systems has attracted intensive research interest (see [40] for a recent survey on this topic). Some existing work, such as UDO [31], also views the query optimizer cost units as tunable knobs. Our view is that the cost units are different from other runtime configuration knobs that change the database server's runtime state. The main impact of the cost units is to influence the query optimizer to generate different query plans.

Autonomous Index Tuning. While some existing work also considers index tuning as part of knob tuning (e.g., UDO [31]), we do think that it falls into another special category of database tuning, just like tuning the cost units, which is worth its own treatment. The reason is similar—index tuning has a different impact on the database system compared to knobs that can change the database server's runtime state. Specifically, index tuning will change the physical data layout of the database, which in some sense is more dramatic as it has broader impact on various database system components, such as metadata management, query optimizer's plan choice, statistics (such as histograms) maintenance, and so on. We refer the readers to [28] for a recent survey on index tuning. The idea of budget-aware index tuning has also been explored recently [32,37], with the similar motivation of constraining the tuning time in practice.

Hyper-parameter Optimization. The HPO problem has been extensively studied in the literature (see [39] for a survey). In addition to simple strategies such as *random search* [5] and bandit-based strategies such as Hyperband [20], many HPO strategies rely on classic Bayesian Optimization (BO), such as Hyperopt [4, 6], SMAC [16,23], Spearmint [29], BOHB [12], and OpenBox [22]. These BO-style HPO strategies view the function from the hyper-parameter values to the ML model quality metric (e.g., accuracy) as a *black box* without utilizing workload properties. While in this paper we use these off-the-shelf HPO strategies without modification, an interesting future direction to explore is to leverage the similarity among the workload queries [10, 27] and use that information to improve BO-style strategies.

6. CONCLUSION

In this paper, we proposed the budget-aware query tuning and workload tuning problems. We highlighted the connection between the query tuning problem and the hyper-parameter optimization (HPO) problem in AutoML, and we proposed solutions based on adapting existing HPO algorithms such as *random search* and *SMAC*. Experimental evaluation shows that (1) query tuning can result in much faster query plans compared to the ones generated by the query optimizer based on the default values of the cost units; and (2) budget-aware workload tuning using simple strategies such as *round robin* or *multi-armed bandit* can significantly reduce the amount of tuning time at workload-level.

We note that the query and workload tuning technologies proposed in this paper are rudimentary as they are simple applications of existing well-known technologies. As a result, they should serve as baseline approaches that future research can reference and compare against. One promising direction for future work, as we briefly mentioned, is to further leverage the similarities among workload queries to improve the BO-style approaches. For instance, a particular set of cost-unit values may be optimal for multiple queries if they share common SQL expressions. As a result, one may want to perform a clustering on the queries and tune each group/cluster of queries as an independent, smaller workload. This can further reduce the tuning time on a large workload, or can have more potential of finding better query plans within a given tuning time budget. Nonetheless, it also raises new challenges such as how to cluster the queries and how to prioritize among the query clusters during workload tuning, which requires further investigation.

Acknowledgement. We thank the anonymous reviewers, Anshuman Dutt, Bailu Ding, and Vivek Narasayya for their valuable feedback on this work.

7. REFERENCES

[1] Postgresql planner cost constants. https://www.postgresql.org/docs/current/runtime-config-query.html, 2024.

[2] P. Auer. Using confidence bounds for exploitation-exploration trade-offs. *J. Mach. Learn. Res.*, 3:397–422, 2002.

[3] P. Auer, N. Cesa-Bianchi, and P. Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2-3):235–256, 2002.

[4] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, 2011.

[5] J. Bergstra and Y. Bengio. Random search for hyper-parameter optimization. *J. Mach. Learn. Res.*, 13:281–305, 2012.

[6] J. Bergstra, D. Yamins, and D. D. Cox. Making a science of model search: Hyperparameter optimization in hundreds of dimensions for vision architectures. In *ICML*, volume 28, pages 115–123, 2013.

[7] N. Bruno, S. Chaudhuri, and R. Ramamurthy. Power hints for query optimization. pages 469–480, 2009.

[8] J. Chang. Sql server query optimizer cost formulas. https://slidetodoc.com/sql-server-query-optimizer-cost-formulas-joe-chang-3/, 2010.

[9] S. Chaudhuri and V. Narasayya. Anytime algorithm of database tuning advisor for microsoft sql server, June 2020.

[10] S. Deep, A. Gruenheid, P. Koutris, J. F. Naughton, and S. Viglas. Comprehensive and efficient workload compression. *Proc. VLDB Endow.*, 14(3):418–430, 2020.

[11] W. Du, R. Krishnamurthy, and M. Shan. Query optimization in a heterogeneous DBMS. In *VLDB*, pages 277–291, 1992.

[12] S. Falkner, A. Klein, and F. Hutter. BOHB: robust and efficient hyperparameter optimization at scale. In *ICML*, volume 80, pages 1436–1445, 2018.

[13] X. He, K. Zhao, and X. Chu. Automl: A survey of the state-of-the-art. *Knowl. Based Syst.*, 212:106622, 2021.

[14] H. Herodotou and S. Babu. Xplus: A sql-tuning-aware query optimizer. *Proc. VLDB Endow.*, 3(1):1149–1160, 2010.

[15] R. Hu, Z. Wang, W. Fan, and S. Agarwal. Cost based optimizer in apache spark 2.2. https://www.databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html, 2017.

[16] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *LION*, volume 6683, pages 507–523, 2011.

[17] L. Kocsis and C. Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[18] V. Leis. Join order benchmark. https://github.com/gregrahn/join-order-benchmark.

[19] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, 2015.

[20] L. Li, K. G. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *J. Mach. Learn. Res.*, 18:185:1–185:52, 2017.

[21] T. Li, J. Zhong, J. Liu, W. Wu, and C. Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *Proc. VLDB Endow.*, 11(5):607–620, 2018.

[22] Y. Li, Y. Shen, W. Zhang, Y. Chen, H. Jiang, M. Liu, J. Jiang, J. Gao, W. Wu, Z. Yang, C. Zhang, and B. Cui. Openbox: A generalized black-box optimization service. In *KDD*, pages 3209–3219, 2021.

[23] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter. SMAC3: A versatile bayesian optimization package for hyperparameter optimization. *JMLR*, 23, 2022.

[24] G. Lohman. Is query optimization a "solved" problem? http://wp.sigmod.org/?p=1075.

[25] G. M. Lohman. The db2 universal database optimizer. https://cs.uwaterloo.ca/ ilyas/CS448W14/ibm.pdf, 2003.

[26] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for distributed queries. In *VLDB*, pages 149–159, 1986.

[27] T. Siddiqui, S. Jo, W. Wu, C. Wang, V. R. Narasayya, and S. Chaudhuri. ISUM: efficiently compressing large and complex workloads for scalable index tuning. In *SIGMOD*, pages 660–673. ACM, 2022.

[28] T. Siddiqui and W. Wu. Ml-powered index tuning: An overview of recent progress and open challenges. *SIGMOD Rec.*, 52(4):19–30, 2023.

[29] J. Snoek, H. Larochelle, and R. P. Adams. Practical bayesian optimization of machine learning algorithms. In *NIPS*, 2012.

[30] C. Wang, Q. Wu, S. Huang, and A. Saied. Economic hyperparameter optimization with blended search strategy. In *ICLR*, 2021.

[31] J. Wang, I. Trummer, and D. Basu. UDO: universal database optimization using reinforcement learning. *Proc. VLDB Endow.*, 14(13):3402–3414, 2021.

[32] X. Wang, W. Wu, C. Wang, V. Narasayya, and S. Chaudhuri. Wii: Dynamic budget reallocation in index tuning. *Proc. ACM Manag. Data*, 2(3), 2024.

[33] Q. Wu, C. Wang, and S. Huang. Frugal optimization for cost-related hyperparameters. In *AAAI*, 2021.

[34] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 6(10):925–936, 2013.

[35] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigümüs, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *ICDE*, pages 1081–1092, 2013.

[36] W. Wu, J. F. Naughton, and H. Singh. Sampling-based query re-optimization. In F. Özcan, G. Koutrika, and S. Madden, editors, *SIGMOD*, pages 1721–1736. ACM, 2016.

[37] W. Wu, C. Wang, T. Siddiqui, J. Wang, V. R. Narasayya, S. Chaudhuri, and P. A. Bernstein. Budget-aware index tuning with reinforcement learning. In Z. G. Ives, A. Bonifati, and A. E. Abbadi, editors, *SIGMOD*, pages 1528–1541, 2022.

[38] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty aware query execution time prediction. *PVLDB*, 7(14):1857–1868, 2014.

[39] T. Yu and H. Zhu. Hyper-parameter optimization: A review of algorithms and applications. *CoRR*, abs/2003.05689, 2020.

[40] X. Zhao, X. Zhou, and G. Li. Automatic database knob tuning: A survey. *IEEE Trans. Knowl. Data Eng.*, 35(12), 2023.