Budget-aware Index Tuning with Reinforcement Learning

Wentao Wu[†], Chi Wang[†], Tarique Siddiqui[†], Junxiong Wang[‡,*]
Vivek Narasayya[†], Surajit Chaudhuri[†], Philip A. Bernstein[†]
[†]Microsoft Research, Redmond [‡]Cornell University
{wentao.wu, wang.chi, tasidd, viveknar, surajitc, philbe}@microsoft.com, jw2544@cornell.edu

ABSTRACT

Index tuning aims to find the optimal index configuration for an input workload. It is a resource-intensive task since it requires making multiple expensive "*what-if*" calls to the query optimizer to estimate the cost of a query given an index configuration without actually building the indexes. In this paper, we study the problem of *budget-aware index tuning* where the number of what-if calls allowed when searching for the optimal configuration during tuning is constrained. This problem is challenging as it requires addressing the trade-off between investing what-if calls on *exploring* new configurations versus *exploiting* a known promising configuration. We formulate budget-aware index tuning as a Markov decision process, and propose a solution based on Monte Carlo tree search, a classic reinforcement learning technology. Experimental evaluation on both standard industry benchmarks and real workloads shows that our solution can significantly outperform alternative budget-aware solutions in terms of the quality of the index configuration.

CCS CONCEPTS

• **Information systems** → **Autonomous database administration**; • **Computing methodologies** → **Reinforcement learning**; • **Computer systems organization** → *Cloud computing*.

KEYWORDS

index tuning, reinforcement learning, budget allocation

ACM Reference Format:
Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, Philip A. Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3514221.3526128

1 INTRODUCTION

Index tuning is an important problem in relational database systems, and has been studied extensively (e.g., [22, 31, 60]). Today's database systems include index tuning advisors that take as input a workload of SQL statements and a set of constraints (e.g., the maximum number of indexes allowed or the total storage taken by the indexes), and recommend an appropriate set of indexes (a.k.a., *configuration*) to create that optimizes the performance of the workload [21, 22, 60]. More recently, with the increasing adoption of database-as-a-service (DBaaS) by enterprises, cloud providers [1, 2, 5, 6] have also started to offer such index tuning capabilities in their platforms [27].

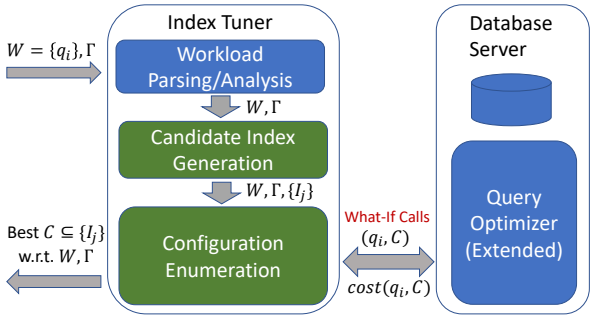


Figure 1: The architecture of cost-based index tuning using what-if optimizer calls, where W is the input workload and $q_i \in W$ is a single query, Γ is a set of tuning constraints, $\{I_j\}$ is the set of candidate indexes generated for W, and $C \subseteq \{I_j\}$ represents an index configuration during enumeration.

A simplified architecture of a typical commercial index tuner is shown in Figure 1. The index tuner contains two major components: (1) *candidate index generation*, which identifies for each query, a set of candidate indexes that can improve the optimizer-estimated cost of the query; and (2) *configuration enumeration*, where the goal is to find an index configuration that minimizes the total cost of the workload. During enumeration, for a configuration C that is considered by the enumeration algorithm, the index tuner consults the query optimizer by calling the "what if" API to estimate the cost of each query in the input workload W when using indexes in C *without* building the indexes [23]. A what-if call invokes the query optimizer and can therefore be resource-intensive. The configuration with the lowest estimated cost found during enumeration that meets the user-specified constraints is returned. A cache is typically used to enable efficient reuse of what-if calls [21]. An end-to-end example of index tuning can be found in Section 2.

Motivation. Index tuning can impose significant demands on resources of the server due to repeated what-if calls to the query optimizer, particularly when tuning complex queries or large workloads. In the cloud database setting, index tuning is often done on the production server [27] since using "B-instances" for each tuned database is prohibitively expensive. Therefore, managing interference from index tuning on production workloads is a challenge. Furthermore, the complexity of the enumeration step is exponential in the number of candidate indexes. It has been shown that restricted versions of the index tuning problem are NP-hard and

*This work was done when Junxiong Wang was at Microsoft Research.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA

ACM ISBN 978-1-4503-9249-5/22/06…$15.00
https://doi.org/10.1145/3514221.3526128

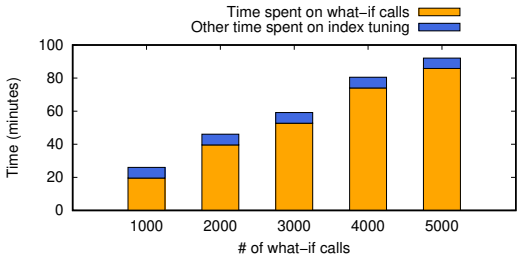


Figure 2: The tuning time of TPC-DS workload when varying the number of what-if calls. The orange bars show the time spent on the what-if calls, whereas the blue bars show the other time spent on index tuning.

even hard to approximate [19, 26]. While prior work has tried to address this by enumerating configurations in *reduced* search spaces (see related work in Section 8), making one what-if call for every configuration and query remains impractical even for such reduced search spaces. For example, the *greedy* algorithm [22] used by state-of-the-art systems [21] would require $O(mnK)$ what-if calls, where m is the number of queries in the input workload, n is the number of candidate indexes, and K is the maximum number of indexes allowed. Not surprisingly, it has been reported [39, 46] that the overhead of what-if calls often dominates the total overheads of index tuning, such as total execution time and CPU/memory resources consumed. As a result, prior work uses techniques to reduce the number of what-if calls, such as making what-if calls only for configurations with certain properties (e.g., the *atomic configurations* in [22]), or smarter reuse of the cached what-if calls [46]. In this paper, we propose a *budget-aware* approach to index tuning, where we constrain the number of what-if calls that are allowed to be made when searching for the best index configuration during the enumeration step. In contrast to prior work, this approach allows the invoker of the index tuning tool to limit the resource demands placed on the server during index tuning.

Budget-aware Configuration Enumeration. We are not the first to highlight the importance of a budget in index tuning. The closest work we are aware of is the *Database Tuning Advisor* (DTA) developed for Microsoft SQL Server, which allows users to specify a maximum budget on the tuning time [3]. While a budget on tuning time is easier for users to specify, budgeting the number of what-if calls has its own merits. First, unlike elapsed time, the number of what-if calls is not sensitive to the effects of the runtime environment, which can be significant, especially in multi-tenant cloud databases. Second, as noted earlier, the time spent on the what-if calls often dominates the index tuning time. Figure 2 illustrates this when running the *greedy* algorithm in a budget-constrained manner (see Section 4) to tune the TPC-DS workload with $K = 20$ recommended indexes. As we can see, what-if calls consistently take around 75% to 93% of the total tuning time across different budgets. [1] Thus, a budget on what-if calls is well correlated with a budget on tuning time. For these reasons, in this paper we use the number of what-if calls as the tuning budget. We also note that the *budget constraint* here is different from other *tuning constraints* that are imposed on the *outcome* of index tuning, such as the maximum number of indexes allowed, the minimum improvement required, etc. This line of *constrained* index tuning has been well studied [18].

Budget Allocation with Reinforcement Learning. The notion of a budget introduces a *budget allocation* problem for index tuning. Specifically, when searching for the best index configuration, we need to decide the configurations and the queries for which to issue what-if calls. This introduces a trade-off between *exploration* and *exploitation*. On one hand, we would like to allocate more what-if calls to configurations that contain known *promising* configurations as subsets; on the other hand, we would also like to *explore* unknown regions in the search space, to increase the chance of finding more promising configurations that have not yet been evaluated. To address this trade-off in a principled manner, we propose a solution based on *reinforcement learning* (RL), by formulating index configuration enumeration/search as a Markov decision process (MDP) [58]. In particular, our solution leverages Monte Carlo tree search (MCTS), a classic RL algorithm [14], and we develop a novel adaptation of the generic MCTS framework in the context of index tuning. MCTS runs a number of simulations (called *episodes*) and we allocate one what-if call in each episode. We have further developed new strategies and policies, tailored for index tuning, to control the overall search and budget allocation process within each episode of MCTS, based on findings from extensive experiments over standard benchmarks and real workloads.

Limitations and Open Issues. This work can be viewed as a first step towards budget-aware index tuning where the focus is limited to the configuration enumeration step. Extending budget-awareness to all components of index tuning remains an open challenge. Also, incorporating the proposed techniques into an index tuning tool (e.g., DTA [21]) requires addressing other important aspects such as ensuring the *anytime* property [21] and being able to handle a user-specified time budget for tuning, which are not the focus of this paper. There has also been recent work on applying RL technologies for *online* index tuning [11, 47, 54], where the index tuner needs to create/drop indexes on the fly to handle workload and database drifts. In contrast, in this paper we limit our focus on the classic *offline* index tuning problem where the workload and database are presumed to be fixed [22, 31, 39, 60]. To this end, our approach does not generalize to a different workload/database.

Summary of Contributions and Paper Overview. To summarize, we have made the following contributions:

- We propose and formalize a novel *budget-aware* index tuning problem that enforces index configuration enumeration to follow a *budget* on the number of what-if calls (Sections 3 and 4).
- We propose a solution to budget-aware configuration enumeration by adapting MCTS, a classic RL technology that has been widely adopted (Sections 5 and 6).
- We conduct extensive experimental evaluation using both industrial benchmark and real workloads (Section 7). The results show that our MCTS-based index tuning algorithm yields significantly better workload cost improvement when running under limited budget, compared to budget-aware variants of the state-of-the-art greedy algorithm and existing RL-based methods [47, 57], in terms of the best configuration found.

[1] We used Microsoft SQL Server 2017 for this measurement. Our observation agrees with previous work [46]. Each what-if call on most TPC-DS queries takes around 1 second since it incurs a full query optimization cycle (e.g., parsing, analyzing, plan enumeration, etc.), and therefore 5,000 what-if calls would take 80 to 100 minutes.

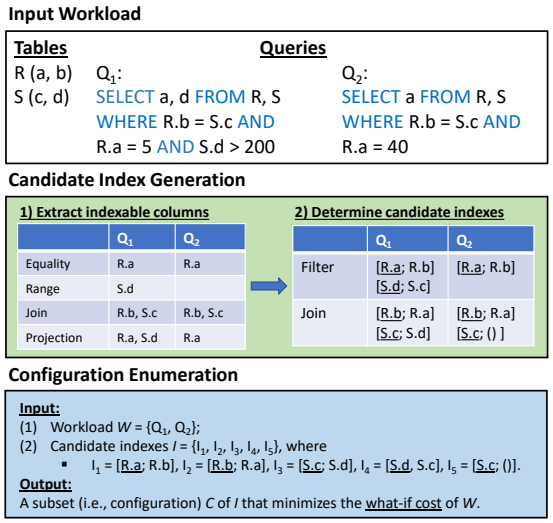


Figure 3: An illustrative example for index tuning.

2 A BRIEF OVERVIEW OF INDEX TUNING

As Figure 1 shows, cost-based index tuning consists of two stages:

- **Candidate index generation** – For each query in the input workload, we generate a set of *candidate indexes* by looking for the *indexable columns* [22]; we take the union as the candidate indexes for the entire workload.
- **Configuration enumeration** – For a given set of candidate indexes, we search for a subset (i.e., *configuration*) of indexes that minimizes the what-if cost of the input workload.

We illustrate these two stages via a concrete example shown in Figure 3. Here, the input workload consists of two queries Q_1 and Q_2. We first look for columns that appear in filter and join predicates appearing in the *where* clause, as well as columns that appear in *group-by* and *order-by* clauses. These columns consist of the basis of the indexable columns. We are also interested in the columns that appear in the projection list of the *select* clause, which can be included as data/payload columns of a *covering index* that can be tremendously useful for "index-only" access paths. We then determine the candidate indexes for each query based on the indexable columns extracted. For simplicity, suppose that we only consider the covering indexes shown in Figure 3, where the *key* columns of the indexes are underscored. We next take a union of these candidate indexes and use them as an input to configuration enumeration, whose goal is to find the best configuration for the workload with the minimum what-if optimizer cost.

3 BUDGET-AWARE INDEX TUNING

In this work, we focus on the *configuration enumeration* component of the index tuning architecture outlined in Figure 1 and we focus on index tuning for data analytic workloads (e.g., TPC-H and TPC-DS). While various implementations have been proposed (e.g., [15, 17, 22]), we focus our discussion on the classic *greedy* search algorithm (Algorithm 1), which has been used by both *AutoAdmin* [22] and *DTA* [21]. A recent empirical study has shown that this greedy algorithm, albeit developed two decades ago, still yields performance improvements comparable to other search algorithms

Algorithm 1: The `Greedy` algorithm for *configuration enumeration* in the index tuning architecture shown in Figure 1.

Input: W, the workload; $\mathcal{I}$, the candidate indexes; K: the cardinality constraint.
Output: $C^{\min}$, the best configuration s.t. $|C^{\min}| \leq K$.

```
C^min ← ∅, cost^min ← cost(W, ∅);
while I ≠ ∅ and |C^min| < K do
    C ← C^min, cost ← cost^min;
    foreach index I ∈ I do
        C_I ← C^min ∪ {I}, cost(W, C_I) ← Σ_{q∈W} cost(q, C_I);
        if cost(W, C_I) < cost then
            C ← C_I, cost ← cost(W, C_I);
    if cost ≥ cost^min then
        return C^min;
    else
        C^min ← C, cost^min ← cost, I ← I − C^min;
return C^min;
```

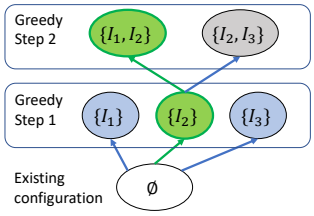


Figure 4: An example of running the greedy algorithm.

used for index tuning [39]. Moreover, we focus on the *cardinality constraint* K on the size of the final index configuration to be returned. However, as we demonstrate in Section 7, it is straightforward to integrate other user-specified tuning constraints, such as the maximum storage space allowed [19], into the budget-aware tuning framework that we develop in this paper.

Example 1 (Greedy). *Figure 4 illustrates the greedy algorithm with three candidate indexes* $\{I_1, I_2, I_3\}$ *and* cardinality constraint $K = 2$. *Here,* $\emptyset$ *represents the* existing index configuration. *In the first greedy step, we consider the three* singleton *configurations* $\{I_1\}$, $\{I_2\}$, *and* $\{I_3\}$. *Suppose that we find that* $\{I_2\}$ *is the best (i.e., with the lowest workload cost). In the second step, we expand* $\{I_2\}$ *by adding one more index, which results in two configurations* $\{I_1, I_2\}$ *and* $\{I_2, I_3\}$. *Suppose that* $\{I_1, I_2\}$ *is better and therefore returned by the greedy algorithm. Note that* $\{I_1, I_3\}$ *is never visited.*

We start by understanding how the greedy algorithm can be adapted in a budget-constrained context. We note that *cost approximation* is unavoidable in this setting as it is infeasible to have one what-if call for every configuration enumerated and every query in the workload. Therefore, we study the implications to the greedy algorithm when combined with cost approximation. Following that, we present a formulation of the budget-aware index tuning problem in this setting where cost approximation has to be used. In Section 4.2, we present example solutions to this formal problem statement that are budget-aware variants of the greedy algorithm.

3.1 Cost Approximation via Derivation

Algorithm 1 does not specify how $\text{cost}(q, C)$, i.e., the cost of a query q given a configuration C, is obtained via approximation approaches

other than using the what-if cost $c(q, C)$ that requires optimizer calls. One common technique, proposed by *AutoAdmin* [22] and also used in *DTA* [21], is the so-called *cost derivation* or, *derived cost*, which we also use in our work. Next, we give an overview of cost derivation and analyze its impact when used with the greedy algorithm in a budget-constrained setting.

3.1.1 Cost Derivation. Given an index configuration C and a query q, the *derived cost* $d(q, C)$ is defined as the *minimum* cost over all subset configurations of C with known what-if costs. Formally,

$$d(q, C) = \min_{S \subseteq C} c(q, S), \tag{1}$$

where $c(q, S)$ stands for the what-if cost of q with the subset S of indexes. This is based on the following assumption on the *monotone* property [34, 56] of index configuration costs:

Assumption 1 (Monotonicity). *Let C_1 and C_2 be two index configurations s.t. $C_1 \subseteq C_2$, and let q be an arbitrary query. Then $c(q, C_2) \leq c(q, C_1)$.*

In other words, including more indexes into a configuration does not increase the what-if cost, assuming that the query optimizer will pick the indexes that lead to the minimum cost of the query. However, Assumption 1 may not always hold, depending on the implementation of the query optimizer's cost model.

Under Assumption 1, it is easy to see that the derived cost is essentially an *upper bound* on the what-if cost, since $c(C, q) \leq \min_{S \subseteq C} c(S, q)$. When the what-if cost $c(q, C)$ is *known*, the derived cost is the same as the what-if cost, i.e., $d(q, C) = c(q, C)$. Therefore, when running the greedy algorithm in a budget-constrained setting, we can simply use $d(q, C)$ for the cost(q, C).

3.1.2 The Greedy Algorithm with Derived Cost. We next study the implications of using derived cost for the greedy algorithm. In general, it is difficult to have optimality guarantees without making assumptions over the cost functions used by the query optimizer [25, 43, 63]. Therefore, in the following we study a special case by focusing on a simpler setting, where we restrict cost derivation over only *singleton* subsets, i.e.,

$$d(q, C) = \min_{z \in C} c(q, \{z\}). \tag{2}$$

Note that $d(q, \{z\}) = c(q, \{z\})$ for any *singleton* configuration $\{z\}$. We use $d(q, \emptyset) = c(q, \emptyset)$ when no (recommended) index is used. We can define the *benefit* of a configuration C given a workload W as

$$b(W, C) = d(W, \emptyset) - d(W, C) = \sum\nolimits_{q \in W} \Big[d(q, \emptyset) - d(q, C) \Big].$$

We prove the following useful property of $b(W, C)$ in [64]:

Theorem 1. *For a given workload W, $b(W, C)$ is a non-negative monotone* submodular *set function if $d(q, C)$ is defined as Equation 2.*

Now, given a workload W with candidate indexes $\mathcal{I}$ and cardinality constraint K, let Ω be all the configurations of $\mathcal{I}$. We can formalize the index selection problem using derived cost as:

$$\max_{S \subseteq \Omega} \{ b(W, C) : |C| \leq K \}.$$

Let C_{opt} be the optimal solution, and let C_{greedy} be the solution by the greedy algorithm. We have the following guarantee [40, 45] based on the submodularity of $b(W, C)$:

Theorem 2. *Given that $b(W, C)$ is a non-negative monotone* submodular *set function by Theorem 1, it follows that*

$$b(W, C_{greedy}) \geq (1 - 1/e) \cdot b(W, C_{opt}).$$

3.2 Budget Allocation in Configuration Search

Given that we have to rely on cost approximation in budget-aware index tuning, we need to decide on which configurations and queries to use what-if calls (and use cost approximation for the others). Specifically, the following *budget allocation* problem arises:

> *Given a budget on the number of what-if calls, how to distribute it to the queries and configurations to achieve the best workload cost improvement?*

We now formulate this budget allocation problem for configuration search by introducing the notion of *budget allocation matrix*.

In our following presentation, let W be a workload, $\mathcal{I}$ be a set of candidate indexes for W, and B be a *budget constraint* on the number of what-if calls.

3.2.1 Budget Allocation Matrix. The budget allocation matrix $\mathcal{B}$ is an $N \times M$ matrix, where $N = 2^{|\mathcal{I}|} - 1$ and $M = |W|$. Each row of $\mathcal{B}$ represents a configuration from indexes in $\mathcal{I}$ and each column of $\mathcal{B}$ represents a query from W. A cell $\mathcal{B}_{ij}$ receives value 1 if the what-if cost $c(q_i, C_j)$ is known (i.e., a what-if call has been made) for the corresponding query $q_i \in W$ and configuration $C_j \in 2^{\mathcal{I}}$ (where $1 \leq i \leq N$ and $1 \leq j \leq M$), and receives value 0 otherwise. We use $v(\mathcal{B}_{ij})$ to represent the value of the cell $\mathcal{B}_{ij}$. Moreover, the sum of all cell values is equal to the budget constraint B, i.e.,

$$\sum\nolimits_{1 \leq i \leq N} \sum\nolimits_{1 \leq j \leq M} v(\mathcal{B}_{ij}) = B. \tag{3}$$

Each specific way of filling in the cell values of $\mathcal{B}$ is called a *layout*:

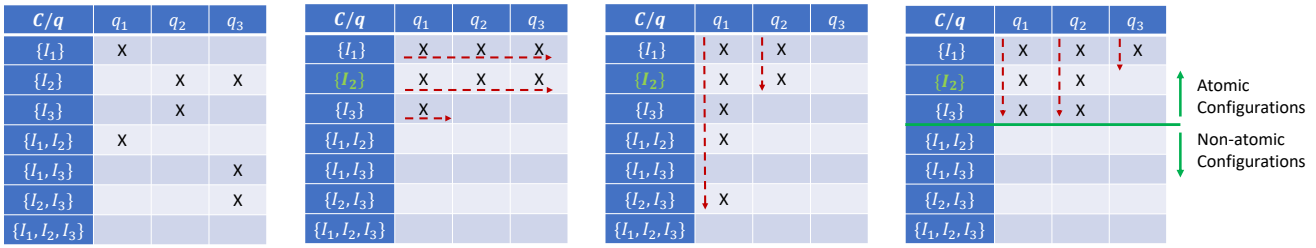
Definition 1 (Layout). *A layout of the budget allocation matrix $\mathcal{B}$ is an ordered mapping $\phi : [B] \rightarrow \{\mathcal{B}_{ij}\}$, where $[B] = \{1, 2, ..., B\}$, $1 \leq i \leq N$, and $1 \leq j \leq M$, such that $v(\phi(b)) = 1$ for $b \in [B]$.*

Intuitively, the *layout* of the budget allocation matrix captures the *trace* of the what-if calls issued during configuration search.

Example 2 (Budget Allocation Matrix). *Figure 5(a) presents an example budget allocation matrix for a workload $W = \{q_1, q_2, q_3\}$ with candidate indexes $\mathcal{I} = \{I_1, I_2, I_3\}$, where the budget constraint is $B = 7$. It also presents the* outcome *of a specific layout, where the cells marked with 'X' receive value 1 and the others receive value 0.*

3.2.2 Budget Allocation as Layout. Let $\mathcal{B}$ be the budget allocation matrix w.r.t. W and $\mathcal{I}$. A configuration search/enumeration algorithm with budget B can use what-if costs for B cells in $\mathcal{B}$, and use derived costs for the others. Each particular way of selecting these B cells corresponds to a particular *layout* of $\mathcal{B}$ — the cells with what-if costs receive value 1 and the others receive value 0. An enumeration algorithm may not visit all cells of $\mathcal{B}$ — the cells not visited receive value 0 by default. The decision on whether to use what-if cost or derived cost can be made during the enumeration.

We now define a *budget allocation* of the configuration enumeration algorithm as a layout L of $\mathcal{B}$. We call such a configuration enumeration algorithm that performs budget allocation a "*budget-aware configuration enumeration* algorithm."



(a) An example (b) First come first serve (c) Two-phase (d) Atomic configuration

Figure 5: Illustration of the budget allocation matrix, where the budget on the what-if calls is $B = 7$.

3.3 Problem Statement

We can now formulate the optimization problem of budget-aware configuration enumeration as follows:

Definition 2 (Optimal Budget-aware Configuration Enumeration). *Given an input workload W with candidate indexes $\mathcal{I}$, a cardinality constraint K, and a budget B on what-if calls, let $\mathcal{B}$ be the budget allocation matrix with respect to W, $\mathcal{I}$, and B, and let $\mathcal{L}$ be all possible layouts of $\mathcal{B}$. Our goal is to find a configuration $C^* \subseteq \mathcal{I}$ s.t. $|C^*| \leq K$ and the derived cost $d(W, C^*)$ is minimized, i.e.,*

$$C^* = \text{argmin}_{C \subseteq \mathcal{I}, |C| \leq K} d(W, C).$$

Here, $d(W, C)$ is restricted to using the what-if costs corresponding to the cells (i.e., configuration-query pairs) in $\mathcal{B}$ with value 1 that are filled by following layouts in $\mathcal{L}$.

This optimal budget-aware configuration enumeration problem is clearly NP-hard, since it contains the original configuration enumeration problem, which is by itself NP-hard [19, 26], as a special case where the budget can be unlimited. In the rest of this paper, we propose various budget-aware configuration enumeration algorithms that are heuristic solutions to this optimization problem. Specifically, in Section 4, we focus on the greedy search algorithm and study how to make it budget-aware; in Section 5, we further propose a novel budget-aware search algorithm based on MCTS.

4 BUDGET-AWARE GREEDY SEARCH

Although there is little prior work on explicit solutions to the problem of budget-aware index tuning, in this section we explore how we can adapt the greedy search algorithm (presented in Algorithm 1) to make it budget-aware. As shown earlier, the greedy algorithm has certain optimality guarantees when combined with cost derivation in a budget-constrained setting (Section 3.1.2). Moreover, as we will discuss shortly, it has an *order-insensitive* property that allows us to focus on the *outcome* of the layouts of the budget allocation matrix without worrying about the orders of filling in the cells (Section 4.1). We leave the investigation of budget-aware variants of other configuration search algorithms as future work.

4.1 Order Insensitivity

One can see that a layout L specifies not only the *outcome* of the budget allocation matrix $\mathcal{B}$ but also the *order* of filling in the cells that lead to the outcome. In general, the order *does* matter, as the behavior of the budget-aware search algorithm (and therefore the configuration returned) can be affected by the matrix $\mathcal{B}$ with *partial* outcome that follows a *prefix* of the layout. This is, however, *not* the case for the greedy algorithm — all layouts that produce the same outcome would yield the same configuration returned by the greedy algorithm. We prove this *order-insensitive* property of the greedy algorithm under a budget-constrained setting in [64]:

Algorithm 2: Two-phase greedy search.

Input: W, the workload; K, the cardinality constraint.
Output: C^{min}: the best index configuration s.t. $|C^{\text{min}}| \leq K$.

```
I ← ∅;
foreach q ∈ W do
    C_q ← Greedy({q}, I_{q}, K);
    I ← I ∪ C_q;
C^min ← Greedy(W, I, K);
return C^min;
```

Theorem 3. *Let L_1 and L_2 be two layouts that produce the same outcome of the matrix $\mathcal{B}$ for the greedy algorithm, and let C_{L_1} and C_{L_2} be the corresponding final configurations returned. Then*

$$d(W, C_{L_1}) = d(W, C_{L_2}).$$

As a result, we can focus on the *outcome* of the layouts without worrying about the order of filling in the matrix cells.

4.2 Budget-aware Greedy Variants

We now discuss different budget allocation strategies to make the greedy algorithm (Algorithm 1) budget-aware.

4.2.1 First Come First Serve. One simple idea is to keep using what-if costs until the budget runs out, after which derived costs are used. This "first come first serve" (FCFS) idea leads to a layout of the budget allocation matrix that fills in the cells in a special *row-major* order, as shown in Figure 5(b). Here we follow Example 1 by assuming that $\{I_2\}$ is the best configuration found in the first greedy step. Clearly, this approach would have trouble when dealing with *large workloads* — it may only be able to visit a couple of *rows* (i.e., configurations) under limited budget. Even for the toy example here, only the first three rows are visited.

4.2.2 Two-phase Search. Another idea is to follow the "two phase" greedy search as was pioneered in *AutoAdmin* [22]. As outlined in Algorithm 2, in the first phase, each query is deemed as a *singleton* workload by itself and tuned by `Greedy` (i.e., Algorithm 1); in the second phase, we take the *union* of the *best indexes* found for each query and use it as the refined set of candidate indexes for the entire workload — we then run `Greedy` again, this time for the entire workload, with this reduced set of candidates.

It is easy to make the two-phase greedy search algorithm "budget-aware" by combining it with the FCFS budget allocation strategy.

This leads to a layout over the budget allocation matrix that fills in the cells in a special *column-major* order in the first phase, as shown in Figure 5(c), before coming back to the *row-major* order in the second phase. The shortcoming of this approach is also clear when dealing with *large search spaces* — it may only be able to visit a couple of *columns* (i.e., queries) under limited budget. Even for the toy example here, only the first two columns are visited.

Special Configurations. One can further focus on allocating budget to only a special subset of the configurations, such as the *single-join atomic configurations* suggested by *AutoAdmin* [22]. Figure 5(d) illustrates how this strategy works when combined with two-phase search and FCFS. Here we only consider atomic configurations of size 1, i.e., singletons. Essentially, it leads to a layout over the budget allocation matrix that fills in the cells in an even special *bounded column-major* order in the first phase, as the filling is *bounded* by the size constraint required by the single-join atomic configurations.

5 BUDGET ALLOCATION WITH MCTS

The budget-aware variants of the greedy algorithm presented in Section 4.2 are based on various rules and heuristics that lead to special classes of layouts of the budget allocation matrix. Hence, they are not able to *adapt* based on the configuration costs observed during search. As was shown in Figure 5, they may have wasted budget on evaluating configurations and queries that turned out to be unnecessary. Essentially, there is a trade-off between *exploitation* and *exploration* when allocating budget what-if calls:

- **Exploitation** – one may want to allocate budget to the configurations that already *show promise*, e.g., configurations that contain the best configuration found by the greedy algorithm so far as a subset;
- **Exploration** – one may also want to allocate budget to the configurations that have *potential*, e.g., configurations that do not contain the best configuration found so far by the greedy algorithm but contain other configurations with similar costs as subsets.

In this section, we propose using reinforcement learning (RL) to make better decisions on this exploitation/exploration trade-off in a principled manner [58]. RL aims for maximizing *cumulative rewards* when taking actions in a search space captured by a Markov decision process (MDP) [13, 51]. In the following, we discuss how we can model configuration search using MDP. We then present a budget-aware search framework based on Monte Carlo tree search (MCTS), a well-known RL technology [14]. In Section 6 we further discuss various design choices and implementation details of the major components in this MCTS-based framework.

5.1 An MDP View of Configuration Search

We model configuration search as an MDP $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, where $\mathcal{S}$ represents the set of *states*, $\mathcal{A}$ represents the set of *actions*, $\mathcal{P}$ represents the set of *transition probabilities*, and $\mathcal{R}$ represents the set of *rewards*. Below we present the details for each of them (see Figure 6 and Example 3 for a concrete example).

5.1.1 States. We define the set of *states* $\mathcal{S}$ of the MDP as all index configurations in the search space. For a given set of candidate indexes $\mathcal{I}$, we have $|\mathcal{S}| = 2^{|\mathcal{I}|}$.

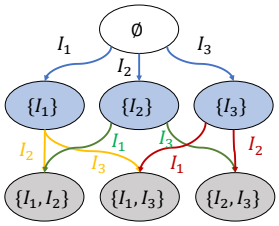


Figure 6: An example of MDP formulation for index tuning.

5.1.2 Actions. For a state $s \in \mathcal{S}$, we define its set of *actions* $\mathcal{A}(s)$ as the indexes that are *not included* by s, i.e., $\mathcal{A}(s) = \mathcal{I} - s$.[2]

5.1.3 Transition Probabilities. Suppose that we are in the state (i.e., configuration) s at time t and we select the action (i.e., index) $a \in \mathcal{A}(s)$, which leads to the state s' at time $t+1$. The transition probabilities $\mathcal{P}$ are then $\Pr(s_{t+1} = s' | s_t = s, a_t = a)$ by the definition of MDP. In our context of index tuning, the next state s' is a deterministic function given s and a, i.e., $s' = f(s, a) = s \cup \{a\}$. As a result, the transition probabilities are $\Pr(s_{t+1} = f(s, a) | s_t = s, a_t = a) = 1$ and 0 otherwise, for all s, a, and t.

5.1.4 Rewards and Expected Returns. The expected *return* of a state s (w.r.t. some policy π) is the expected *cumulative* reward when starting from s and following the policy π.[3] The expected return is well-known as the *state-value function* in the RL literature, denoted as $V_\pi(s)$. Similarly, one can define the *action value function* $Q_\pi(s, a)$, which indicates the expected return when starting at s, taking the action a, and then following the policy π. The goal of RL is to find an *optimal* policy π^* that *maximizes* the state-value function. We denote this optimal state-value function as $V^*(s)$. Interestingly, π^* also maximizes the action-value function, denoted by $Q^*(s, a)$ [58]. As a result, to find π^*, one can maximize either $V_\pi(s)$ or $Q_\pi(s, a)$.

Since the goal of RL is to maximize the expected return (as encoded by the state/action value function), the semantics of the expected return has to be consistent with the optimization goal of configuration search. Therefore, we define the expected return of a state s as the expected *percentage improvement* of configurations that contain s as a subset. Specifically, the percentage improvement of a configuration C over the workload W is

$$\eta(W, C) = \left(1 - \frac{\text{cost}(W, C)}{\text{cost}(W, \emptyset)}\right) \times 100\%, \tag{4}$$

where $\text{cost}(W, C) = \sum_{q \in W} \text{cost}(q, C)$. In budget-aware configuration search, $\text{cost}(W, C)$ may not be evaluated precisely by using the what-if costs, due to budget constraints. As a result, we use the derived cost $d(W, C) = \sum_{q \in W} d(q, C)$ as an approximation.

Example 3. *Figure 6 presents an example of the MDP for index configuration search. Again, we follow Example 1 using a search space with candidate indexes $\{I_1, I_2, I_3\}$ and cardinality constraint $K = 2$. Here, we only show state transitions with nonzero probabilities, and we have omitted the rewards to avoid clutter. One cannot further expand the states with two indexes by adding more indexes, as otherwise the cardinality constraint would be violated. In this paper, we call such states without outgoing transitions* terminal states.

[2] We abuse notation by using s to also denote the configuration that it represents.
[3] A policy $\pi(a|s)$ specifies the action a that should be taken when at the state s. A policy can be either deterministic or stochastic.

5.2 MCTS for Budget-aware Index Tuning

When the state space $\mathcal{S}$ is small, one can just use *dynamic programming* to find $V^*(s)$ or $Q^*(s,a)$, given the recursive relationships between the state/action values [58]. Unfortunately, dynamic programming is infeasible for a large state/action space, encountered when tuning large workloads — the computation time and amount of memory required to run dynamic programming to compute $V^\pi(s)$ or $Q^\pi(s,a)$ would be enormous. As a result, one has to seek approximate solutions for such MDP problems. We next show how we adapt Monte Carlo tree search (MCTS), a popular RL technology that does not require explicitly representing the entire state/action space [14], to address this scalability challenge.

At a high level, MCTS is a simulation-based RL technology that estimates the action value function $Q(s,a)$ using sampled traces by following the state transitions in the MDP. However, MCTS does not need to estimate $Q(s,a)$ for *all* state-action pairs (s,a); rather, it only focuses on state-action pairs that show promise in the long run. It achieves this by *progressively* pruning unpromising pairs and narrowing down the search space.

Specifically, MCTS organizes the entire search space using a *tree* structure (see Figure 7 and Example 4 for a concrete example). Each node in the tree represents a state (i.e., configuration) in the MDP, and each outgoing edge of a node represents an action (i.e., the next index to be included) of the state. MCTS then runs a number of *episodes* that progressively expand the search tree and sample configurations. In each episode we start from the *root* of the tree and execute the following steps as depicted in Figure 7:

- **Selection** – We pick an outgoing edge (i.e., an action) from the current node (i.e., the current state) based on some *action selection policy* (Section 6.1), and then visit the corresponding child node (i.e., the next state) by following the selected edge;
- **Expansion** – If the current node is a *leaf* that has been visited, we *expand* the search tree by following the edge selected and adding the node that represents the next state;
- **Simulation** – After reaching a leaf that does not represent a terminal state and has not been visited yet, we perform a *rollout* by randomly adding indexes into the configuration represented by the leaf (Section 6.2);
- **Update** – We evaluate the configuration returned by the rollout to obtain its what-if cost, and use that to update the *average reward* of *every* node along the *path* from the root to the leaf.

Algorithm 3 presents a formal description of this framework when applied to the MDP problem for configuration search. Below we outline the details of some functions that have been omitted:

- `CreateNode` – It takes a configuration C and the candidate indexes $\mathcal{I}$ as inputs, and creates a new node in the search tree that represents the state s corresponding to C. Meanwhile, it initializes the action set $\mathcal{A}(s)$ of s, which includes all candidate indexes that are not covered by C, i.e., $\mathcal{A}(s) = \mathcal{I} - C$. Moreover, it initializes bookkeeping information for each action $a \in \mathcal{A}(s)$ w.r.t. the action selection policy (Section 6.1).
- `EvaluateCostWithBudget` – This is where the budget is allocated. Our current strategy is to pick one query q and use its what-if cost for $\text{cost}(q,C)$. For the rest of the workload we simply use cost derivation, i.e., $\text{cost}(W,C) \leftarrow c(q,C) + \sum_{q' \in W, q' \neq q} d(q',C)$. We currently pick the query q with probability proportional to its derived cost, though other strategies are possible. The remaining budget b' is simply $b' \leftarrow b - 1$.
- `GetOrCreateNextState` – It retrieves the node corresponding to the next state s' based on the given state s and action a. If s' does not exist in the search tree, it creates a node for s' by calling `CreateNode`$(s', \mathcal{I})$.
- `SelectAction` – It selects an action $a \in \mathcal{A}(s)$ for the given state s, based on the action selection policy (Section 6.1).
- `Rollout` – It generates a configuration by randomly inserting indexes. The details of this procedure, called a *rollout policy*, again depend on the action selection policy (Section 6.2).
- `ExtractBestConfiguration` – There are various ways of extracting the best configurations from the search tree, which again depend on the action selection policy (Section 6.3).

Algorithm 3: MCTS for budget-aware index tuning.

Input: W, the workload; $\mathcal{I}$, the candidate indexes; K: the cardinality constraint; B, the budget.
Output: $C^{\min}$, the best index configuration s.t. $|C^{\min}| \leq K$.

```
Main:
s_0 ← CreateNode(∅, I), b ← B;
while b > 0 do
    b ← RunEpisode(s_0, b, W, K);
C^min ← ExtractBestConfiguration(s_0);
return C^min;

RunEpisode(s: state, b: budget, W, K)
C ← SampleConfiguration(s, K);
(cost(W,C), b') ← EvaluateCostWithBudget(W, C, b);
η(W,C) ← (1 − cost(W,C)/cost(W,∅)) × 100%;
foreach state s' in the path that leads to C do
    Update the average reward of s' with η(W,C);
return b';

SampleConfiguration(s: state, K)
if s is leaf then
    if s has not been visited before then
        return Rollout(s);
    else if s is a terminal state, i.e., |s| == K then
        return s;
a ← SelectAction(s);
s' ← GetOrCreateNextState(s, a);
return SampleConfiguration(s', K);
```

Example 4 (MCTS for index configuration enumeration). *Figure 7 illustrates the four major steps in one episode of MCTS for configuration search. Once again, we follow Example 1 by assuming a search space with candidate indexes* $\{I_1, I_2, I_3\}$*. We start from the root and pick* I_2 *based on the action selection policy (i.e., the* selection *step). Since* $\{I_2\}$ *is a leaf that has been visited before, we expand it by picking the action* I_3*, which leads to adding the new (leaf) state* $\{I_2, I_3\}$ *into the search tree (i.e., the* expansion *step). We then (recursively) visit this new leaf. Since it has not been visited before, we run a rollout to estimate its average return (i.e., the* simulation *step). We finally update the average returns for the states/actions along the path from the root to the leaf (i.e.,* $\{I_2\}$ *and* $\{I_2, I_3\}$*), with the reward observed on the sampled configuration by rollout (i.e., the* update *step).*

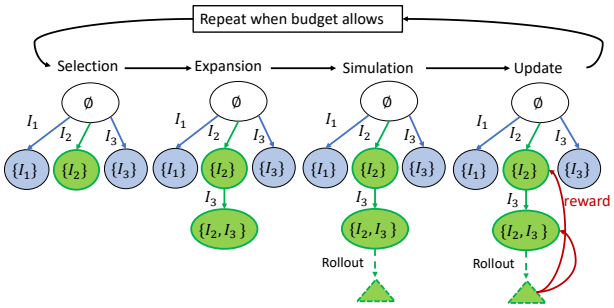


Figure 7: MCTS for index configuration enumeration.

6 IMPLEMENTATIONS OF MCTS POLICIES

As we have noted in Algorithm 3, various policies can be employed for (1) action selection, (2) rollout, and (3) extraction of the best configuration. In this section, we present design considerations and implementation details of these MCTS policies.

6.1 Action Selection Policy

The action selection policy $\pi(a|s)$ specifies that, given a state s, which action a should be taken next. Below we discuss two popular policies, *UCT* and *ϵ-greedy* that we adapted for index tuning.

6.1.1 UCT. One common action selection policy used in practice for MCTS is UCT [38], which models action selection as a stochastic multi-armed bandit problem and uses the *upper confidence bound* (UCB) for the trade-off between *exploration* and *exploitation* [8–10]. Specifically, let s be a state and $\mathcal{A}(s)$ be its action set. Let $N(s)$ be the number of visits to the state s, and $n(s,a)$ be the number of visits to an action $a \in \mathcal{A}(s)$. Clearly, $N(s) = \sum_{a\in\mathcal{A}(s)} n(s,a)$. Assuming that all rewards are in the range of $[0,1]$, which is true in our MDP formulation of index configuration search since we use percentage improvements as rewards (ref. Equation 4), the *UCT* policy picks the action $a \in \mathcal{A}(s)$ that maximizes the UCB score, i.e.,

$$\operatorname*{argmax}_a \left[\hat{Q}(s,a) + \lambda \cdot \sqrt{\frac{\ln N(s)}{n(s,a)}} \right]. \tag{5}$$

Here, we use $\hat{Q}(s,a)$ to represent the current estimate of the action value function $Q(s,a)$, which in our context represents the *average* return by taking the action a at the state s. The initial value of $\hat{Q}(s,a)$ is zero, since no reward has been accumulated. λ is a constant that balances *exploration* and *exploitation*. We chose $\lambda = \sqrt{2}$ as suggested by the original UCT paper [38].

One problem of UCT is that it makes slow progress when there is a large number of candidate indexes. Based on the UCB1 criterion (i.e., Equation 5), once a tree node is expanded, all of its child nodes have to be visited *at least once* before any of them can be further expanded, because child nodes that are not visited, i.e., with $n(s,a) = 0$ in Equation 5, would receive *infinite* UCB score. As a result, given a relatively small budget B, only the first one or two levels of the search tree would be expanded.

6.1.2 ϵ-Greedy. Another well-known policy is ϵ-greedy [58], where one picks the *best* action found so far, i.e., with the largest $\hat{Q}(s,a)$ as in Equation 5, with probability $1-\epsilon$, and picks an action *uniformly* randomly from the rest with probability $\frac{\epsilon}{|\mathcal{A}(s)|-1}$. One apparent drawback is that ϵ-greedy does not distinguish among the remaining actions except for the current best one. That is, it is equally likely to select the *next-to-best* action and the *worst* action. As a result, when applied to MCTS, people usually use some variants of ϵ-greedy. For example, one popular variant is the *Boltzmann's exploration* [48], which chooses an action a with probability $\Pr(a|s) = \frac{e^{\hat{Q}(s,a)/\tau}}{\sum_{b\in\mathcal{A}(s)} e^{\hat{Q}(s,b)/\tau}}$. Here, $\tau > 0$ is some *temperature parameter* that needs to be set.

Motivated by the Boltzmann's exploration but to get rid of the dependency on additional hyperparameters, we propose a simpler variant of ϵ-greedy that is more intuitive in our context. That is, we simply pick an action a with probability that is proportional to its estimated action value:

$$\Pr(a|s) = \frac{\hat{Q}(s,a)}{\sum_{b\in\mathcal{A}(s)} \hat{Q}(s,b)}. \tag{6}$$

We also note that all these variants of ϵ-greedy do not change the convergence of MCTS in the long run. That is, given enough episodes such that all states are visited sufficient number of times, the estimated action value function $\hat{Q}(s,a)$ can eventually converge to the actual action value function $Q(s,a)$. In fact, even choosing an action just uniformly randomly can guarantee the convergence of MCTS [48]. Nonetheless, when MCTS is run under limited budget (as in our case), different action selection policies can make a difference due to their different convergence rates. In our evaluation we observed that our ϵ-greedy variant works well, which often outperforms UCT significantly under the budget constraints we tested (see [64]), though UCT is theoretically more attractive due to its provable convergence rate in the long run [38].

One new challenge raised by our ϵ-greedy variant under limited budget is that the estimates $\hat{Q}(s,a)$ are typically very "sparse." Clearly, to make $\hat{Q}(s,a)$ a reasonable estimate, one has to take the action a at least once. In practical index tuning applications with budget constraints, it is often infeasible to visit *every* action $b \in \mathcal{A}(s)$ given the large action space, as we have pointed out in the UCT case. To address this challenge, we assign some "prior reward" to each action a that has not been taken yet. Specifically, we define the prior reward as the percentage improvement of the *singleton* configuration corresponding to a, i.e., $\eta(\mathcal{W}, \{a\})$. We then initialize $\hat{Q}(s,a)$ with $\eta(\mathcal{W}, \{a\})$, *independent* of the state s. This idea, however, introduces a nontrivial problem of computing percentage improvements for singleton configurations *under limited budget*, as it requires what-if calls. Algorithm 4 outlines our current approach. In a budget-constrained setting, the basic idea is to pick singletons *selectively* to evaluate. Specifically, for each budget what-if call, we first select a query q (via `QuerySelection`) and then select one of its candidate indexes I that has not been evaluated yet (via `IndexSelection`), based on certain query selection and index selection policies that we will discuss shortly.

Another question is how much budget B' on the number of what-if calls we should give to Algorithm 4. In our current implementation, we set $B' = \min(B/2, P)$, where B is the total budget for configuration search and P is the total number of *query-index pairs*, which works well in our evaluation. It is an interesting question regarding the optimal setting of B', which we leave for future work.

Query Selection. Once again, there can be various strategies for query selection. One idea is to use the same strategy as in the `EvaluateCostWithBudget` procedure of Algorithm 3. However,

Algorithm 4: Compute percentage improvements for singleton configurations under limited budget

Input: W, the workload; $\mathcal{I}$, the candidate indexes; B', the budget on the number of what-if calls.
Output: $\eta(W, \{I\})$ for all $I \in \mathcal{I}$.

```
foreach I ∈ I do
    cost(W, {I}) ← c(W, ∅);
for 1 ≤ b ≤ B' do
    q ← QuerySelection(W);
    I ← IndexSelection(q, I);
    cost(W, {I}) ← cost(W, {I}) − c(q, ∅) + c(q, I);
foreach I ∈ I do
    η(W, {I}) ← (1 − cost(W,{I}) / cost(W,∅)) × 100%;
```

our goal here is different. Specifically, we aim for finding impactful indexes (i.e., singleton configurations) at workload-level, which implies that we should encourage more on *exploring* new queries (so that we can find new indexes) than exploiting queries that have been selected before. Hence, in our current implementation, we use the simple *round-robin* strategy, which is robust and works well on the workloads that we tested in our experiments (Section 7). Of course, this is not a perfect solution (e.g., it may give more than necessary what-if calls to unimportant queries), and perhaps cannot scale to extreme cases (e.g., if the workload size is larger than the budget on what-if calls). To address the scalability issue, we can further combine round-robin with other heuristics. For example, one can select a subset (e.g., a random sample) of the queries and apply round-robin only within this subset.

Index Selection. In our current implementation, we choose to favor candidate indexes over large tables, since for a particular query intuitively indexes over large tables are more useful. That is, we first select indexes on the largest table accessed by the query, and then select indexes on the second largest table, and so on. This strategy works well since we focus on cardinality constraint. Other strategies can be devised if we want to further optimize for other constraints (e.g., storage space), which we leave for future work.

6.2 Rollout Policy

The rollout policy specifies how to generate a configuration by *randomly* inserting indexes. Specifically, we first generate a "look-ahead step size" $l \in \{0, 1, ..., K - d\}$ in a uniformly random manner, where d is the *depth* of the current state s in the search tree.

Depending on the action selection policy, we then insert l indexes, randomly chosen from $\mathcal{A}(s)$, as follows:

- *UCT* – we choose the l indexes *uniformly*;
- *ϵ-greedy* – we choose the l indexes from $\mathcal{A}(s)$ w.r.t. the priors of actions, i.e., the probability of sampling $a \in \mathcal{A}(s)$ is given by Equation 6 where we use the prior reward of a for $\hat{Q}(s, a)$.

Clearly, there are other possible rollout policies. For instance, rather than using a randomly generated look-ahead step size, one can use a fixed look-ahead step size. The $\hat{Q}(s, a)$ obtained by using such a policy is no longer an *unbiased* estimator of $Q(s, a)$. However, it may make sense in a budget-limited setting where we have to resort to approximations such as derived costs after running out of budget. For example, in the above strategy, one may choose to be more "myopic" by focusing on a small step size, say, 0 or 1, which would allow for more exploration in the neighborhood of the current state compared to remote regions in the entire search space.[4] We compare the standard rollout policy and its "myopic" variants in more detail in the full version of this paper [64].

6.3 Extraction of the Best Configuration

Finally, one needs to extract the best configuration from the search tree. Again, there are various strategies, and we have considered the following in our implementation:

- *Best Configuration Explored* (BCE), where we simply return the best configuration found during MCTS, among the configurations *explored*. The explored configurations include all configurations corresponding to states in the final expanded search tree, as well as configurations sampled by rollouts.
- *Best Greedy* (BG), where we use a greedy strategy to traverse the search tree. There are various variants based on what metric to be optimized, which we will discuss next.

The *BG* strategy here works similarly as the `Greedy` procedure in Algorithm 1, though it is not tied to minimizing the workload cost. For example, one can pick the action that maximizes the estimated average return, i.e., $\hat{Q}(s, a)$, in each greedy step. Or, if *UCT* is used as the action selection policy, then one can pick the action that maximizes the UCB score, or even the most frequent action.

We currently choose to implement the *BG* strategy by just reusing Algorithm 1, for a couple of considerations. First, it does not rely on the action selection policy used. Second, it improves code reusability and makes the integration with existing index tuning systems (such as *DTA* [21]) easier. Third, it significantly outperforms the *BCE* strategy, as reported by our evaluation results (see [64]). Fourth, by Theorem 2 (see Section 3.1.2) and Theorem 3 (see Section 4.1), it has certain merits and desired properties when combined with cost derivation and budget allocation strategies.

Remark. We note that, a similar greedy strategy, which picks the action a in each greedy step that maximizes $\hat{Q}(s, a)$, is in theory the optimal policy [58], *if* $\hat{Q}(s, a)$ has converged to the *optimal* $Q(s, a)$, i.e., the $Q^*(s, a)$ in Section 5.2. In practice when the budget is typically small compared to the size of the search space, this convergence condition rarely holds. As a result, this variant of the *BG* strategy does not have any optimality guarantee in general.

7 EXPERIMENTAL EVALUATION

We now report experimental results on evaluating the performance of our budget-aware configuration search algorithm based on MCTS. In our experiments, we vary the budget on the number of what-if calls, and measure the percentage improvement, i.e., Equation 4, of a given workload in terms of the *actual* what-if cost. We also vary the cardinality constraint K, i.e., the maximum number of indexes to be returned, to test the robustness of the algorithm.

Datasets and Workloads. We used various benchmark and real workloads in our study. Table 1 summarizes the information of the workloads. For benchmark workloads, we use the join order benchmark (**JOB**) proposed by Leis et al. [43] that is publicly available at [42], as well as both the **TPC-H** and **TPC-DS** benchmarks with scaling factor 10.[5] We further use two real workloads, denoted

[4] A step size of 0 means that we only pick the configuration represented by the current state without inserting additional indexes.
[5] **JOB** contains 113 query instances in total, which are grouped into 33 templates, and we pick one query instance from each template. We also follow the same protocol for

Name	Size	# Queries	# Tables	Avg. # Joins	Avg. # Filters	Avg. # Scans
JOB	9.2GB	33	21	7.9	2.5	8.9
TPC-H	sf=10	22	8	2.8	0.3	3.7
TPC-DS	sf=10	99	24	7.7	0.5	8.8
Real-D	587GB	32	7,912	15.6	0.2	17
Real-M	26GB	317	474	20.2	1.5	21.7

Table 1: Summary of database and workload statistics.

by **Real-D** and **Real-M** in Table 1, which are significantly more complicated compared to the benchmark workloads, in terms of schema complexity (e.g., the number of tables), query complexity (e.g., the average number of joins and table scans contained by a query), and database/workload size.

Experimental Setup. In our experiments, we varied the cardinality constraint K within {5, 10, 20}. Since the two workloads **JOB** and **TPC-H** are relatively small, we varied the budget B on the number of what-if calls within {50, 100, 200, 500, 1000}; on the other hand, for the remaining three large workloads, **TPC-DS**, **Real-D**, and **Real-M**, we varied B from 1,000 to 5,000, with 1,000 increment. We perform all experiments using Microsoft SQL Server 2017 under Windows Server 2019, running on a workstation equipped with 2.6 GHz Intel CPUs and 192 GB main memory. Since our MCTS-based approach requires randomization, we run it five times with different seeds for the random number generator (RNG) it uses, and we report its average performance as well as the observed standard deviation (shown as error bars).

Remark. Due to space constraints, we defer the experimental results on the two small workloads, **JOB** and **TPC-H**, to [64]. The observations on these two workloads are similar.

7.1 Comparison with Baselines

We first compare our MCTS-based algorithm with the following baseline algorithms that are budget-aware greedy variants of Algorithm 1, discussed in Section 4.2:

- *Vanilla greedy*, which simply runs greedy search at workload-level, i.e., the one-phase framework outlined in Algorithm 1, with FCFS as its budget allocation strategy.
- *Two-phase greedy*, which runs greedy search at both query-level and workload-level, i.e., the two-phase framework in Algorithm 2, with FCFS as its budget allocation strategy.
- *AutoAdmin greedy*, which uses the same two-phase greedy search framework as *two-phase greedy*, with the exception that it focuses on allocating budget to only *atomic* configurations suggested by [22], again in an FCFS manner.

Figures 8 to 10 present the results over various workloads. In each figure, the x-axis represents the number of what-if calls allowed (as well as the corresponding tuning time in minutes), and the y-axis represents the percentage improvement on the overall workload cost. Here we show the results for our MCTS algorithm based on the following setting: (1) ϵ-greedy for action selection, (2) myopic rollout with step size 0 (i.e., no random indexes added), and (3) greedy extraction of the best configuration (i.e., the *BG* strategy in Section 6.3). This setting overall gives the best and most consistent performance among the various settings we tested in our ablation study that can be found in the full version of this paper [64].

7.1.1 TPC-DS Results. Figure 8 presents the **TPC-DS** results. Our MCTS-based approach can consistently outperform the baseline approaches. Among the baselines, *vanilla greedy* performs the worst overall, whereas *two-phase greedy* and *AutoAdmin greedy* perform similarly. The gap between *vanilla greedy* and the other two is often quite large, especially when the budget is small, though *vanilla greedy* can eventually catch up with increased budget.

7.1.2 Real-D Results. Our approach again significantly outperforms the baseline approaches on **Real-D**, especially when the budget is small. For example, with 1,000 what-if calls, the improvements obtained by our approach and the baseline approaches are 60% vs. 40%, respectively, for all $K \in \{5, 10, 20\}$. Meanwhile, although the three baselines perform similarly, *vanilla greedy* finally outperforms the other two approaches with 10% more improvement when setting $K = 10$ and $K = 20$ with 5,000 what-if calls.

7.1.3 Real-M Results. On **Real-M**, our approach significantly outperforms all the baselines when $K = 10$ and $K = 20$, while it performs better than or closely to *two-phase greedy* and *AutoAdmin greedy* when $K = 5$. *Vanilla greedy*, however, performs significantly worse. The improvement from *vanilla greedy* is 0% when the number of budget what-if calls is set to 1,000 and never exceeds 5% when increasing the budget from 1,000 to 5,000, for all $K \in \{5, 10, 20\}$. Meanwhile, our approach reports improvement around 35% to 40%, a 7× to 8× relative improvement.

7.1.4 Summary. Based on the results above, we observe that (1) MCTS outperforms the baselines consistently and often significantly, especially with small tuning budget; and (2) there is no clear winner among the three baselines.

7.2 Comparison with Existing RL Approaches

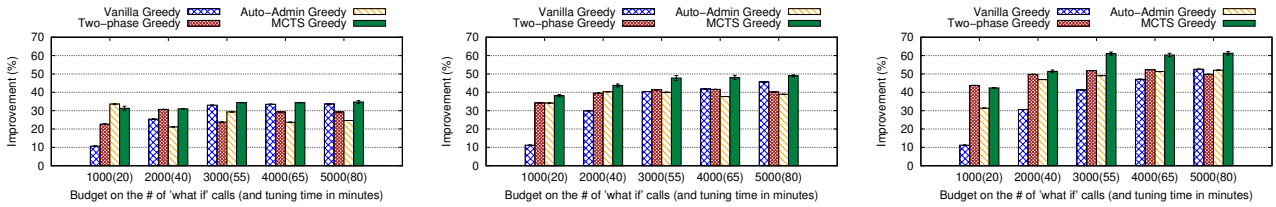
We next compare our MCTS-based approach with two existing RL approaches: (1) *DBA bandits* [47] and (2) *No DBA* [57]. Figures 11 to 13 summarize the results on **TPC-DS**, **Real-D**, and **Real-M**, where we present the improvements of the *best* configurations found by *DBA bandits* and *No DBA* in each experimental setting.

7.2.1 DBA Bandits. The idea is to model the index selection problem using *contextual combinatorial bandit* [50]. We focus on the "static workload" setting described in [47], where we break the overall execution into multiple rounds. In each round, we make a what-if call to each query in the workload to observe its cost under the current index configuration selected by *DBA bandits*, and we use this information to compute the rewards required by *DBA bandits* to refine its index recommendations. [6]

Our MCTS-based approach significantly outperforms *DBA bandits* on all three workloads (with up to 3.3× relative improvement). While *DBA bandits* can quickly land on an initial configuration that looks promising, it makes slow progress on the follow-up refinements. To demonstrate this, Figure 14 presents the percentage improvement of the best configuration found by *DBA bandits* in each round. One reason for this behavior could be the large search spaces for the workloads used in our evaluation, which consist of

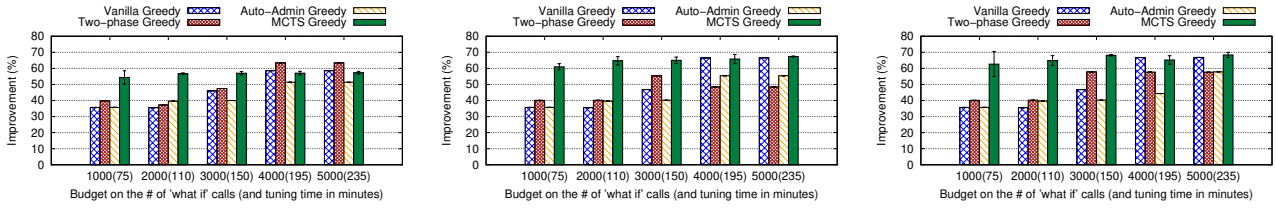
both **TPC-H** and **TPC-DS**. While the multi-instance versions of the workloads are also interesting, various other techniques can be used for workload compression [20, 29], which introduces another dimension that can affect the performance of index tuning. For these reasons, we leave the study of multi-instance workloads as future work.

[6] *DBA bandits* targets an *online* index tuning scenario where actual execution time is used to compute rewards. In our *offline* index tuning setting, we do not actually execute queries and that is why we have to use what-if cost instead.



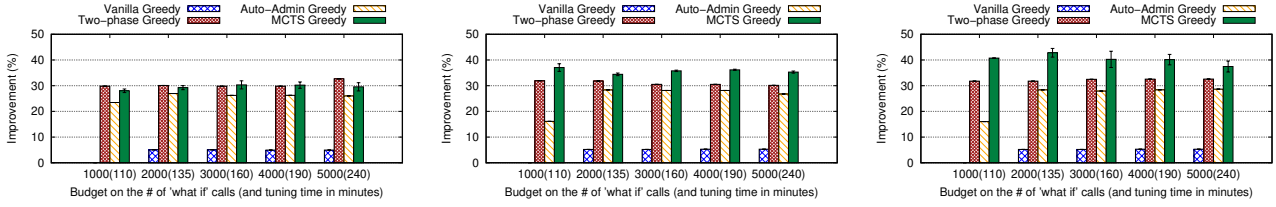
(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 8: End-to-end performance comparison on TPC-DS with budget-aware `Greedy` variants.



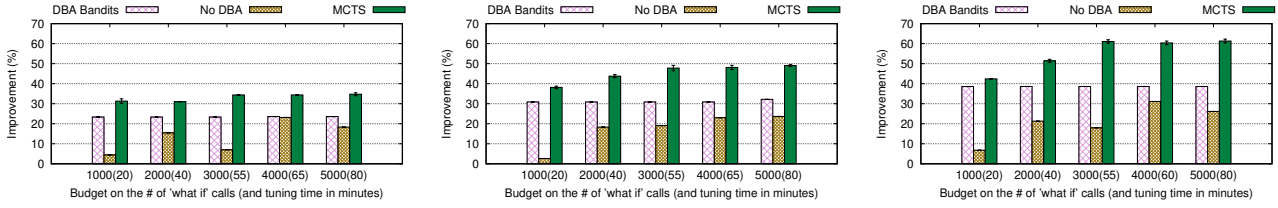
(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 9: End-to-end performance comparison on Real-D with budget-aware `Greedy` variants.



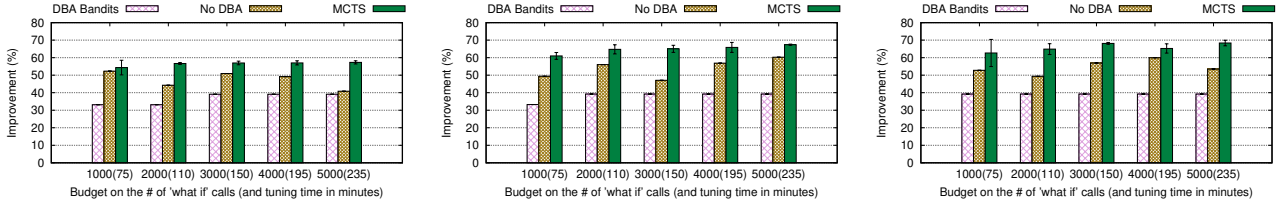
(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 10: End-to-end performance comparison on Real-M with budget-aware `Greedy` variants.



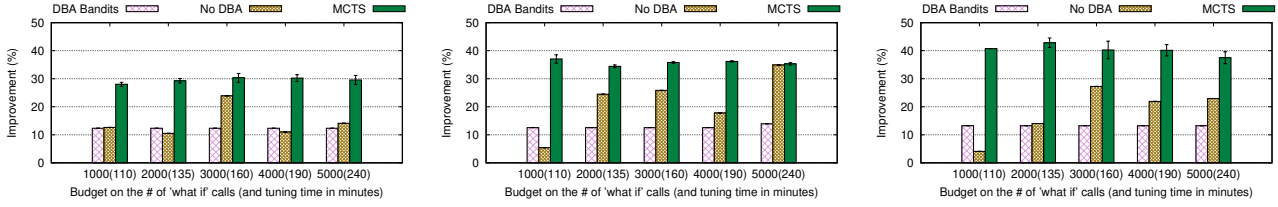
(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 11: End-to-end performance comparison on TPC-DS with existing RL approaches.



(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 12: End-to-end performance comparison on Real-D with existing RL approaches.



(a) $K = 5$ (b) $K = 10$ (c) $K = 20$

Figure 13: End-to-end performance comparison on Real-M with with existing RL approaches.

hundreds to thousands of candidate indexes. Under limited budget, only a few index configurations in this large search space can be explored by *DBA bandits*. This highlights the importance of prioritizing queries and candidate indexes when it comes to allocating the budget of what-if calls, which has been discussed in Section 6.1 in detail. On the other hand, *DBA bandits* leveraged certain featurization techniques to represent candidate indexes, which merits further consideration in our MCTS-based approach. In our evaluation, we observed that appropriate featurization could help identify promising index configurations more quickly (e.g., see the results on **TPC-H** and **JOB** in [64]).

7.2.2 No DBA. The idea is to use deep RL to solve the index tuning problem. As noted in [39], *No DBA* is not suitable for offline index tuning in general without adaptation, as it only supports a subset of TPC-H queries and can only recommend single-column indexes. It is also much slower compared to other offline index tuning algorithms, as it uses actual query execution time to compute rewards and needs GPU support in general for training deep neural networks (DNNs). We therefore implement a variant of *No DBA* with the following adaptations: (1) we use *one-hot* encoding h_C to represent an index configuration/state $C \subseteq \mathcal{I}$, where $\mathcal{I}$ represents the universe of candidate indexes; we set $h_C[i] = 1$ if $i \in \mathcal{I}$ appears in C, and 0 otherwise; (2) we use optimizer's estimated what-if cost instead of query execution time to compute rewards; (3) we use deep Q-learning [35] as our deep RL technology; (4) we use a relatively small DNN that contains three fully connected layers, each with 96 neurons, and we use `relu` as the activation function.[7] Following [39], we only use CPU for training the DNN, for a fair comparison against the other technologies.

We ran our variant of *No DBA* in a budget-constrained manner, where again we split the overall execution into multiple rounds and in each round we evaluate each query in the workload for the current configuration selected by deep Q-learning to compute its reward. From the results shown in Figures 11 to 13, our MCTS-based approach significantly outperforms *No DBA* in most of the cases that we tested (with up to 14.4× relative improvement). One important issue of *No DBA* we observed when running under limited budget is again its slow convergence, as demonstrated in Figure 14.

7.3 Comparison with DTA

We further compare our MCTS-based approach against *DTA* [3, 21], since DTA allows user to specify an (optional) budget on the tuning time in minutes. However, DTA cannot accept the number of what-if calls as budget. Therefore, we make a best-effort comparison to DTA by giving it the same amount of tuning time that our MCTS-based approach spends. Since DTA supports storage constraint (SC), we compare our approach and DTA with and without SC. When enabling SC, we set the allowed storage size as 3× of the database size, which is the default setting used by DTA [3].

We note that this comparison is not completely fair. First, DTA needs to deal with a more challenging problem of allocating the time budget to all components of index tuning, not just the configuration enumeration step, to ensure the *anytime* property [21] that our current work does not consider. Second, DTA is a full-fledged index tuning tool with additional optimizations (e.g., "table subset" selection [7, 21], index merging [24], etc.) that we did not implement. Our purpose of comparing with DTA here is to understand if, by only improving the configuration enumeration algorithm via budget-aware tuning, what the performance gap is from a comprehensive index tuning tool that also takes a time budget as input. An interesting question is how much improvement we would see if we integrate our techniques into DTA's configuration enumeration step, which is beyond the scope of this paper.

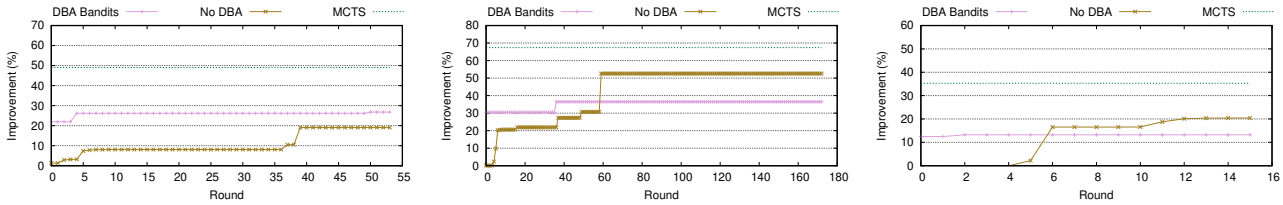
Figure 15 summarizes the results on **TPC-DS**, **Real-D**, and **Real-M**. Here we do not show the variances of our approach when using different RNG seeds, which are similar to those shown in Figures 8 to 13, to avoid clutter. On **TPC-DS**, our approach achieves similar performance compared to DTA, except for a couple of cases where DTA's performance drops. For example, as shown in Figure 15(d), when disabling SC and setting $K = 10$, our approach outperforms DTA with a budget of 1,000 what-if calls – 40% vs. 10% improvement. On **Real-D**, we observe some different patterns compared to what we observed on **TPC-DS**. For example, as shown in Figure 15(e), when disabling SC and setting $K \in \{5, 10\}$, our approach performs quite similarly to DTA; however, when setting $K = 20$, our approach outperforms DTA. We can see that DTA's performance even drops before finally catching up, when increasing its tuning time limit. Such non-monotonic behavior of DTA has also been observed in previous work [29]. Figure 15(b) shows another example when enabling SC and setting $K = 20$, where DTA could not return useful indexes in several cases. DTA adopts a time-slice based architecture [21]. In each time slice, it consumes the next batch of queries, and the recommended indexes are based on the queries tuned so far. Internally, DTA uses a cost-based priority queue to select the next query to tune [21]. It is possible that DTA hits some costly query and spends the time budget tuning that query without finding useful indexes. Our approach overcomes such issues with stochastic policies on selecting queries and configurations. DTA may alleviate such issues, too, when given sufficient time. On **Real-M**, we can outperform DTA in some cases and DTA outperforms our approach in some other cases. For example, as shown in Figure 15(f), when disabling SC and setting $K = 10$, with 3,000 what-if calls our approach reports 35% improvement while DTA reports 0% improvement (i.e., no indexes recommended). Overall, DTA's behavior is again non-monotonic.

Furthermore, considering the impact of the storage constraint on our MCTS-based approach, we observe that increasing the storage space in general allows our approach to find better configurations, which is intuitive and consistent with findings on other index configuration search algorithms [39].

8 RELATED WORK

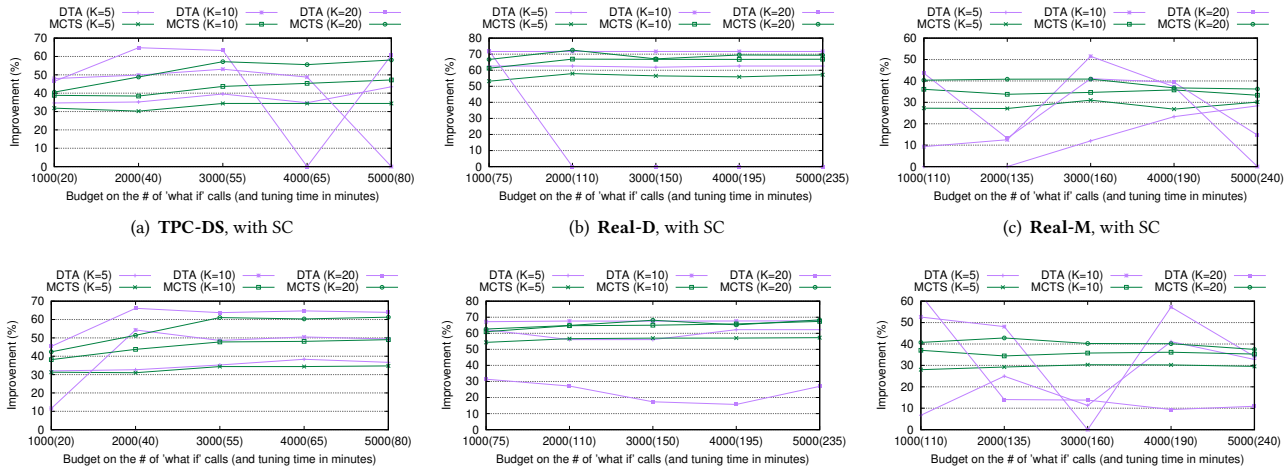
The problem of index tuning has been studied extensively in the literature. Existing work includes, but not limited to, the following: *Drop* [62], *AutoAdmin* [22], *DTA* [21], *DB2Advisor*, *Relaxation* [15], *CoPhy* [28], *Dexter* [37], *Extend* [55], etc. We refer the readers to the recent work [39] for more details of these solutions, which also conducted a benchmark study that compares their performances. Based on the reported results, *DTA* with the greedy search algorithm can yield the state-of-the-art performance. This is one important

[7] The original implementation of *No DBA*, available at [4], allows for various types of deep-RL agents, such as deep Q-learning (DQN), SARSA [53], Cross Entropy Method (CEM) [52], and Deep Deterministic Policy Gradient (DDPG) [44], and it does not specify which particular agent to use.



(a) **TPC-DS**, $K = 10$ (b) **Real-D**, $K = 10$ (c) **Real-M**, $K = 20$

Figure 14: Convergence of *DBA bandits* and *No DBA*. The budget on the number of what-if calls is set to 5,000. For comparison, we also include the average improvement reported by our MCTS-based approach in each chart.



(a) **TPC-DS**, with SC (b) **Real-D**, with SC (c) **Real-M**, with SC

(d) **TPC-DS**, without SC (e) **Real-D**, without SC (f) **Real-M**, without SC

Figure 15: End-to-end performance comparison vs. DTA with and without storage constraint (SC).

reason for us to focus on the greedy algorithm when considering the budget-aware index tuning problem.

There has also been recent work on applying ML/RL [11, 47, 54] to *online* index tuning [16], which is different from the *offline* setting studied in this paper. To the best of our knowledge, there is limited work on applying ML/RL to offline index tuning. Ding et al. [30] proposed to train a classifier using execution data to improve configuration search, with the (different) goal of reducing the chance of query performance regression after building the indexes selected by the index tuner. *No DBA* [57] used deep RL and real query execution time to compute reward. As reported by [39], it took *No DBA* eight hours to tune a workload of ten queries on the 1GB TPC-H database and came up with "on par" results compared to traditional approaches. Recent work by Lan et al. [41] proposed a deep-RL based index advisor similar to *No DBA* [57], though using what-if cost for reward calculation. Again, it could tune a subset of TPC-H queries at 1GB scale with improvement similar to traditional approaches; however, the tuning time was not reported. Nevertheless, none of these efforts has considered the budget-constrained index configuration search problem that we studied in this paper.

Compared to using the number of what-if calls as budget, using execution time may be more attractive from user's perspective. Indeed, it is not our intention to expose the number of what-if calls as a tunable knob to the end user — we propose to retain the same control that *DTA* provides today, which is tuning time as a budget [3]. Internally, we can map this time budget to the number of what-if calls allowed (e.g., we can divide the time budget by the average time of a what-if call), which is transparent to the end user.

Our choice of using MCTS for index tuning was motivated by SkinnerDB [59], which used MCTS to select the optimal join order. Compared to other RL techniques such as policy iteration [12], value iteration [49], and Q-learning [36, 61], the tree structure leveraged by MCTS captures the hierarchical relationship between index configurations naturally. Further optimizations are possible. For example, one may consider the "rapid action value estimation" (RAVE) method [33] in the update policy. One may also consider combining featurization technologies, such as ones used by *DBA bandits* [47] or other deep RL [32, 35] approaches, with our MCTS framework to achieve more compact representation of the state/action space.

9 CONCLUSION

In this paper, we studied the budget-aware index configuration enumeration/search problem. We presented a formal problem definition of budget allocation in configuration search, as well as analysis when combining budget allocation with the classic greedy search algorithm. We further proposed a solution based on MCTS, by modeling configuration search as an MDP. Our evaluation demonstrates that the MCTS-based solution can often significantly outperform the budget-aware greedy algorithms as well as variants of existing RL-based approaches to index tuning, across a variety of industrial benchmarks and real workloads.

Acknowledgments: We thank the anonymous reviewers, Arnd Christian König, and Bailu Ding for their valuable feedback.

REFERENCES

[1] [n.d.]. Amazon Relational Database Service. https://aws.amazon.com/rds/.
[2] [n.d.]. Azure SQL Database. https://azure.microsoft.com/en-us/products/azure-sql/database/.
[3] [n.d.]. DTA utility. https://docs.microsoft.com/en-us/sql/tools/dta/dta-utility?view=sql-server-ver15.
[4] [n.d.]. GitHub Repository of *No DBA*. https://github.com/shankur/autoindex.
[5] [n.d.]. Google Cloud SQL. https://cloud.google.com/sql.
[6] [n.d.]. Oracle Database Cloud Service. https://www.oracle.com/database/.
[7] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*. 496–505.
[8] Jean-Yves Audibert, Rémi Munos, and Csaba Szepesvari. 2006. Use of variance estimation in the multi-armed bandit problem. (2006).
[9] Peter Auer. 2002. Using Confidence Bounds for Exploitation-Exploration Trade-offs. *J. Mach. Learn. Res.* 3 (2002), 397–422.
[10] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. Finite-time Analysis of the Multiarmed Bandit Problem. *Mach. Learn.* 47, 2-3 (2002), 235–256.
[11] Debabrota Basu, Qian Lin, Weidong Chen, Hoang Tam Vo, Zihong Yuan, Pierre Senellart, and Stéphane Bressan. 2015. Cost-Model Oblivious Database Tuning with Reinforcement Learning. In *DEXA*. 253–268.
[12] R. E. Bellman. 1957. *Dynamic Programming*. Princeton University Press.
[13] Dimitri P. Bertsekas. 2005. *Dynamic programming and optimal control, 3rd Edition*. Athena Scientific.
[14] Cameron Browne, Edward Jack Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez Liebana, Spyridon Samothrakis, and Simon Colton. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Trans. Comput. Intell. AI Games* 4, 1 (2012), 1–43.
[15] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic Physical Database Tuning: A Relaxation-based Approach. In *SIGMOD*. 227–238.
[16] Nicolas Bruno and Surajit Chaudhuri. 2007. An Online Approach to Physical Design Tuning. In *ICDE*. 826–835.
[17] Nicolas Bruno and Surajit Chaudhuri. 2007. Physical design refinement: The 'merge-reduce' approach. *ACM Trans. Database Syst.* 32, 4 (2007), 28.
[18] Nicolas Bruno and Surajit Chaudhuri. 2008. Constrained physical design tuning. *Proc. VLDB Endow.* 1, 1 (2008), 4–15.
[19] Surajit Chaudhuri, Mayur Datar, and Vivek R. Narasayya. 2004. Index Selection for Databases: A Hardness Study and a Principled Heuristic Solution. *IEEE Trans. Knowl. Data Eng.* 16, 11 (2004), 1313–1323.
[20] Surajit Chaudhuri, Ashish Kumar Gupta, and Vivek R. Narasayya. 2002. Compressing SQL workloads. In *SIGMOD*. 488–499.
[21] Surajit Chaudhuri and Vivek Narasayya. 2020. Anytime Algorithm of Database Tuning Advisor for Microsoft SQL Server.
[22] Surajit Chaudhuri and Vivek R. Narasayya. 1997. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*. 146–155.
[23] Surajit Chaudhuri and Vivek R. Narasayya. 1998. AutoAdmin 'What-if' Index Analysis Utility. In *SIGMOD*. 367–378.
[24] Surajit Chaudhuri and Vivek R. Narasayya. 1999. Index Merging. In *ICDE*.
[25] Sunil Choenni, Henk M. Blanken, and Thiel Chang. 1993. On the Selection of Secondary Indices in Relational Databases. *Data Knowl. Eng.* 11, 3 (1993).
[26] Douglas Comer. 1978. The Difficulty of Optimum Index Selection. *ACM Trans. Database Syst.* 3, 4 (1978), 440–445.
[27] Sudipto Das, Miroslav Grbic, Igor Ilic, Isidora Jovandic, Andrija Jovanovic, Vivek R. Narasayya, Miodrag Radulovic, Maja Stikic, Gaoxiang Xu, and Surajit Chaudhuri. 2019. Automatically Indexing Millions of Databases in Microsoft Azure SQL Database. In *SIGMOD*. 666–679.
[28] Debabrata Dash, Neoklis Polyzotis, and Anastasia Ailamaki. 2011. CoPhy: A Scalable, Portable, and Interactive Index Advisor for Large Workloads. *Proc. VLDB Endow.* 4, 6 (2011), 362–372.
[29] Shaleen Deep, Anja Gruenheid, Paraschos Koutris, Jeffrey F. Naughton, and Stratis Viglas. 2020. Comprehensive and Efficient Workload Compression. *Proc. VLDB Endow.* 14, 3 (2020), 418–430.
[30] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R. Narasayya. 2019. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. In *SIGMOD*. 1241–1258.
[31] S. J. Finkelstein, M. Schkolnick, and P. Tiberio. 1988. Physical Database Design for Relational Databases. *ACM Trans. Database Syst.* 13, 1 (1988).
[32] Vincent François-Lavet, Peter Henderson, Riashat Islam, Marc G. Bellemare, and Joelle Pineau. 2018. An Introduction to Deep Reinforcement Learning. *Found. Trends Mach. Learn.* 11, 3-4 (2018), 219–354.
[33] Sylvain Gelly and David Silver. 2011. Monte-Carlo tree search and rapid action value estimation in computer Go. *Artif. Intell.* 175, 11 (2011), 1856–1875.
[34] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. 1997. Index Selection for OLAP. In *ICDE*. 208–219.
[35] Todd Hester, Matej Vecerík, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Dan Horgan, John Quan, Andrew Sendonaris, Ian Osband, Gabriel Dulac-Arnold, John P. Agapiou, Joel Z. Leibo, and Audrunas Gruslys. 2018. Deep Q-learning From Demonstrations. In *AAAI*. 3223–3230.
[36] Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. 1996. Reinforcement Learning: A Survey. *J. Artif. Intell. Res.* 4 (1996), 237–285.
[37] Andrew Kane. 2017. Introducing Dexter, the Automatic Indexer for Postgres. https://medium.com/@ankane/introducing-dexter-the-automatic-indexer-for-postgres-5f8fa8b28f27.
[38] Levente Kocsis and Csaba Szepesvári. 2006. Bandit Based Monte-Carlo Planning. In *ECML*. 282–293.
[39] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? An Experimental Evaluation of Index Selection Algorithms. *Proc. VLDB Endow.* 13, 11 (2020), 2382–2395.
[40] Andreas Krause and Daniel Golovin. 2014. Submodular Function Maximization. In *Tractability: Practical Approaches to Hard Problems*. Cambridge University Press, 71–104.
[41] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An Index Advisor Using Deep Reinforcement Learning. In *CIKM*. 2105–2108.
[42] Viktor Leis. [n.d.]. Join Order Benchmark. https://github.com/gregrahn/join-order-benchmark.
[43] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215.
[44] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *ICLR*.
[45] G. L. Nemhauser et al. 1978. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.* 14, 1 (1978).
[46] Stratos Papadomanolakis, Debabrata Dash, and Anastassia Ailamaki. 2007. Efficient Use of the Query Optimizer for Automated Database Design. ACM.
[47] Malinga Perera, Bastian Oetomo, Benjamin I. P. Rubinstein, and Renata Borovica-Gajic. 2020. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. *CoRR* abs/2010.09208 (2020).
[48] Laurent Péret and Frédérick Garcia. 2004. On-Line Search for Solving Markov Decision Processes via Heuristic Sampling. In *ECAI*. 530–534.
[49] M. L. Puterman and M. C. Shin. 1978. Modified policy iteration algorithms for discounted Markov decision problems. *Management Science* 24, 11 (1978).
[50] Lijing Qin, Shouyuan Chen, and Xiaoyan Zhu. 2014. Contextual Combinatorial Bandit and its Application on Diversified Online Recommendation. In *SDM*. 461–469.
[51] S. Ross. 2014. *Introduction to stochastic dynamic programming*. Academic press.
[52] Reuven Rubinstein. 1999. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability* 1, 2 (1999), 127–190.
[53] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge.
[54] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online Index Selection Using Deep Reinforcement Learning for a Cluster Database. In *ICDE Workshops*.
[55] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient Scalable Multi-attribute Index Selection Using Recursive Strategies. In *ICDE*. 1238–1249.
[56] Karl Schnaitter, Neoklis Polyzotis, and Lise Getoor. 2009. Index Interactions in Physical Design Tuning: Modeling, Analysis, and Applications. *Proc. VLDB Endow.* 2, 1 (2009), 1234–1245.
[57] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The Case for Automatic Database Administration using Deep Reinforcement Learning. *CoRR* abs/1801.05643 (2018).
[58] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[59] Immanuel Trummer, Junxiong Wang, Deepak Maram, Samuel Moseley, Saehan Jo, and Joseph Antonakakis. 2019. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. In *SIGMOD*. 1153–1170.
[60] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. 2000. DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes. In *ICDE*. 101–110.
[61] Christopher J. C. H. Watkins and Peter Dayan. 1992. Technical Note Q-Learning. *Mach. Learn.* 8 (1992), 279–292.
[62] Kyu-Young Whang. 1985. Index Selection in Relational Databases. In *Foundations of Data Organization*. 487–500.
[63] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *ICDE*. 1081–1092.
[64] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A. Bernstein. 2022. *Budget-aware Index Tuning with Reinforcement Learning (Extended Version)*. Technical Report. Microsoft Research. https://www.microsoft.com/en-us/research/people/wentwu/publications/