Wii: Dynamic Budget Reallocation In Index Tuning

XIAOYING WANG, Simon Fraser University, Canada
WENTAO WU, Microsoft Research, USA
CHI WANG, Microsoft Research, USA
VIVEK NARASAYYA, Microsoft Research, USA
SURAJIT CHAUDHURI, Microsoft Research, USA

Index tuning aims to find the optimal index configuration for an input workload. It is often a time-consuming and resource-intensive process, largely attributed to the huge amount of “what-if” calls made to the query optimizer during configuration enumeration. Therefore, in practice it is desirable to set a budget constraint that limits the number of what-if calls allowed. This yields a new problem of budget allocation, namely, deciding on which query-configuration pairs (QCP’s) to issue what-if calls. Unfortunately, optimal budget allocation is NP-hard, and budget allocation decisions made by existing solutions can be inferior. In particular, many of the what-if calls allocated by using existing solutions are devoted to QCP’s whose what-if costs can be approximated by using cost derivation, a well-known technique that is computationally much more efficient and has been adopted by commercial index tuning software. This results in considerable waste of the budget, as these what-if calls are unnecessary. In this paper, we propose “Wii,” a lightweight mechanism that aims to avoid such spurious what-if calls. It can be seamlessly integrated with existing configuration enumeration algorithms. Experimental evaluation on top of both standard industrial benchmarks and real workloads demonstrates that Wii can eliminate significant number of spurious what-if calls. Moreover, by reallocating the saved budget to QCP’s where cost derivation is less accurate, existing algorithms can be significantly improved in terms of the final configuration found.

CCS Concepts: • Information systems → Query optimization; Autonomous database administration.

Additional Key Words and Phrases: Index tuning, Budget allocation, What-if API, Query optimization

ACM Reference Format:

1 INTRODUCTION

Index tuning aims to find the optimal index configuration (i.e., a set of indexes) for an input workload of SQL queries. It is often a time-consuming and resource-intensive process for large and complex workloads in practice. From user’s perspective, it is therefore desirable to constrain the index tuner/advisor by limiting its execution time and resource, with the compromise that the goal of index tuning shifts to seeking the best configuration within the given time and resource constraints. Indeed, commercial index tuning engines, such as the Database Tuning Advisor (DTA) developed for

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 2836-6573/2024/6-ART182
https://doi.org/10.1145/3654985
Fig. 1. The architecture of budget-aware index tuning with "Wii", i.e., what-if (call) interception, where \( W \) represents the input workload, \( q_i \in W \) represents an individual SQL query in the workload, \( \Gamma \) represents a set of tuning constraints, \( B \) represents the budget on the number of what-if calls allowed. Moreover, \( \{z_j\} \) represents the set of candidate indexes generated for \( W \), and \( C \subseteq \{z_j\} \) represents an index configuration proposed during configuration enumeration.

Microsoft SQL Server, have been offering a timeout option that allows user to explicitly control the execution time of index tuning to prevent it from running indefinitely [1, 7]. More recently, there has been a proposal of budget-aware index tuning that puts a budget constraint on the number of “what-if” (optimizer) calls [46], motivated by the observation that most of the time and resource in index tuning is spent on what-if calls [19, 26] made to the query optimizer during configuration enumeration (see Figure 1).

A what-if call takes as input a query-configuration pair (QCP) and returns the estimated cost of the query by utilizing the indexes in the configuration. It is the same as a regular query optimizer call except for that it also takes hypothetical indexes, i.e., indexes that are proposed by the index tuner but have not been materialized, into consideration [9, 40]. There can be thousands or even millions of potential what-if calls when tuning large and complex workloads [36]. Therefore, it is not feasible to make a what-if call for every QCP encountered in configuration enumeration/search. As a result, one key problem in budget-aware index tuning is budget allocation, where one needs to determine which QCP’s to make what-if calls for so that the index tuner can find the best index configuration. Unfortunately, optimal budget allocation is NP-hard [6, 11, 46]. Existing budget-aware configuration search algorithms [46] range from adaptations of the classic greedy search algorithm [8] to more sophisticated approaches with Monte Carlo tree search (MCTS) [18], which allocate budget by leveraging various heuristics. For example, the greedy-search variants adopt a simple “first come first serve” (FCFS) strategy where what-if calls are allocated on demand, and the MCTS-based approach considers the rewards observed in previous budget allocation steps to decide the next allocation step. These budget allocation strategies can be inferior. In particular, we find in practice that many of the what-if calls made are unnecessary, as their corresponding what-if costs are close to the approximations given by a well-known technique called cost derivation [8]. Compared to making a what-if call, cost derivation is computationally much more efficient and has been integrated into commercial index tuning software such as DTA [1, 7]. In the rest of this paper, we refer to the approximation given by cost derivation as the derived cost. Figure 2 presents the distribution of the relative gap between what-if cost and derived cost when tuning the TPC-DS benchmark workload with 99 complex queries. We observe that 80% to 90% of the what-if calls were made for QCP’s with relative gap below 5%, for two state-of-the-art budget-aware configuration search algorithms two-phase greedy and MCTS (Section 2.2). If we know that the derived cost is
indeed a good approximation, we can avoid such a spurious what-if call. The challenge, however, is that we need to learn this fact before the what-if call is made. The best knowledge we have so far is that, under mild assumption on the monotonicity of query optimizer’s cost function (i.e., a larger configuration with more indexes should not increase the query execution cost), the derived cost acts as an upper bound of the what-if cost (Section 2.2.2). However, the what-if cost can still lie anywhere between zero and the derived cost. In this paper, we take one step further by proposing a generic framework that develops a lower bound for the what-if cost. The gap between the lower bound and the upper bound (i.e., the derived cost) therefore measures the closeness between the what-if cost and the derived cost. As a result, it is safe to avoid a what-if call when this gap is small and use the derived cost as a surrogate.

Albeit a natural idea, there are a couple of key requirements to make it relevant in practice. First, the lower bound needs to be nontrivial, i.e., it needs to be as close to the what-if cost as possible—an example of a trivial but perhaps useless lower bound would be always setting it to zero. Second, the lower bound needs to be computationally efficient compared to making a what-if call. Third, the lower bound needs to be integratable with existing budget-aware configuration enumeration algorithms. In this paper, we address these requirements as follows.

Nontriviality. We develop a lower bound that depends only on common properties of the cost functions used by the query optimizer, such as monotonicity and submodularity, which have been widely assumed by previous work [10, 15, 22, 31, 44] and independently verified in our own experiments [41]. In a nutshell, it looks into the marginal cost improvement (MCI) that each individual index in the given configuration can achieve, and then establishes an upper bound on the cost improvement (and therefore a lower bound on the what-if cost) of the given configuration by summing up the upper bounds on the MCI’s of individual indexes (Section 3.1). We further propose optimization techniques to refine the lower bound for budget-aware greedy search algorithms (Section 4.1) and MCTS-based algorithms (Section 4.2).

Efficiency. We demonstrate that the computation time of our lower bound is orders of magnitude less compared to a what-if call, though it is in general more expensive than computing the upper bound, i.e., the derived cost (Section 6.4). For example, as shown in Figure 16(b), when running the MCTS configuration enumeration algorithm on top of the TPC-DS benchmark, on average it takes 0.02 ms and 0.04 ms to compute the derived cost and our lower bound, respectively; in contrast, the average time of making a what-if call to the query optimizer is around 800 ms.

Integratability. We demonstrate that our lower bound can be seamlessly integrated with existing budget-aware index tuning algorithms (Section 5). From a software engineering perspective, the
integration is non-intrusive, meaning that there is no need to change the architecture of the current cost-based index tuning software stack. As illustrated in Figure 1, we encapsulate the lower-bound computation inside a component called “Wii,” which is shorthand for “what-if (call) interception.” During configuration enumeration, Wii intercepts every what-if call made to the query optimizer, computes the lower bound of the what-if cost, and then checks the closeness between the lower bound and the derived cost (i.e., the upper bound) with a confidence-based mechanism (Section 3.3). If Wii feels confident enough, it will skip the what-if call and instead send the derived cost back to the configuration enumerator.

More importantly, we demonstrate the efficacy of Wii in terms of (1) the number of what-if calls it allows to skip (Section 6.3) and (2) the end-to-end improvement on the final index configuration found (Section 6.2). The latter is perhaps the most valuable benefit of Wii in practice, and we show that, by reallocating the saved budget to what-if calls where Wii is less confident, it can yield significant improvement on both standard industrial benchmarks and real customer workloads (Section 6.2). For example, as showcased in Figure 6(f), with 5,000 what-if calls as budget and 20 as the maximum configuration size allowed, on TPC-DS Wii improves the baseline two-phase greedy configuration enumeration algorithm by increasing the percentage improvement of the final configuration found from 50% to 65%; this is achieved by skipping around 18,000 unnecessary what-if calls, as shown in Figure 14(b).

Last but not least, while we focus on budget-aware index tuning in this paper, Wii can also be used in a special situation where one does not enforce a budget on the index tuner, namely, the tuner has unlimited budget on the number of what-if calls. This special situation may make sense if, for example, one has a relatively small workload. Wii plays a different role here. Since there is no budget constraint, Wii cannot improve the quality of the final configuration found, as the best quality can anyways be achieved by keeping on issuing what-if calls to the query optimizer. Instead, by skipping spurious what-if calls, Wii can significantly improve the overall efficiency of index tuning. For example, without a budget constraint, when tuning the standard TPC-H benchmark with 22 queries, Wii can reduce index tuning time by 4× while achieving the same quality on the best configuration found (Section 6.8).

2 PRELIMINARIES
In this section, we present a brief overview of the budget-aware index configuration search problem.

2.1 Cost-based Index Tuning
As Figure 1 shows, cost-based index tuning consists of two stages:

- **Candidate index generation.** We generate a set of candidate indexes for each query in the workload based on the indexable columns [8]. Indexable columns are those that appear in the selection, join, group-by, and order-by expressions of a SQL query, which are used as key columns for fast seek-based index look-ups. We then take the union of the candidate indexes from individual queries as the candidate indexes for the entire workload.

- **Configuration enumeration.** We search for a subset (i.e., a configuration) of the candidate indexes that can minimize the what-if cost of the workload, with respect to constraints such as the maximum number of indexes allowed or the total amount of storage taken by the index configuration.

Index tuning is time-consuming and resource-intensive, due to the large amount of what-if calls issued to the query optimizer during configuration enumeration/search. Therefore, previous work proposes putting a budget on the amount of what-if calls that can be issued during configuration search [46]. We next present this budget-aware configuration search problem in more detail.
2.2 Budget-aware Configuration Search

2.2.1 Problem Statement. Given an input workload $W$ with a set of candidate indexes $I$, a set of constraints $\Gamma$, and a budget $B$ on the number of what-if calls allowed during configuration enumeration, our goal is to find a configuration $C^* \subseteq I$ whose what-if cost $c(W, C^*)$ is minimized under the constraints given by $\Gamma$ and $B$.

In this paper, we focus on index tuning for data analytic workloads $W$ (e.g., the TPC-H and TPC-DS benchmark workloads). Although the constraints in $\Gamma$ can be arbitrary, we focus on the cardinality constraint $K$ that specifies the maximum configuration size (i.e., the number of indexes contained by the configuration) allowed. Moreover, under a limited budget $B$, it is often impossible to know the what-if cost of every query-configuration pair (QCP) encountered during configuration enumeration. Therefore, to estimate the costs for QCP’s where what-if calls are not allocated, one has to rely on approximation of the what-if cost without invoking the query optimizer. One common approximation technique is cost derivation [7, 8], as we discuss below.

2.2.2 Cost Derivation. Given a QCP $(q, C)$, its derived cost $d(q, C)$ is the minimum cost over all subset configurations of $C$ with known what-if costs. Formally,

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

Here, $c(q, S)$ is the what-if cost of $q$ using only a subset $S$ of indexes from the configuration $C$.

We assume the following monotone property [15, 31] of index configuration costs w.r.t. to an arbitrary query $q$:

**Assumption 1 (Monotonicity).** Let $C_1$ and $C_2$ be two index configurations where $C_1 \subseteq C_2$. Then $c(q, C_2) \leq c(q, C_1)$.

That is, including more indexes into a configuration does not increase the what-if cost. Our validation results using Microsoft SQL Server show that monotonicity holds with probability between 0.95 and 0.99, on a variety of benchmark and real workloads (see [41] for details). Under Assumption 1, we have

$$d(q, C) \geq c(q, C),$$

i.e., derived cost is an upper bound $U(q, C)$ of what-if cost:

$$U(q, C) = d(q, C) = \min_{S \subseteq C} c(q, S).$$

2.2.3 Existing Solutions. The budget-aware configuration search problem is NP-hard. At the core of this problem is budget allocation, namely, to decide on which QCP’s to make what-if calls. Existing heuristic solutions to the problem include: (1) vanilla greedy, (2) two-phase greedy, (3) AutoAdmin greedy, and (4) MCTS. Since (2) and (3) are similar, we omit (3) in this paper.
Vanilla greedy. Figure 3(a) illustrates the vanilla greedy algorithm with an example of three candidate indexes \(\{z_1, z_2, z_3\}\) and the cardinality constraint \(K = 2\). Throughout this paper, we use \(\emptyset\) to represent the existing configuration. Vanilla greedy works step-by-step, where each step adopts a greedy policy to choose the next index to be included that can minimize the workload cost on the chosen configuration. In this example, we have two greedy steps. The first step examines the three singleton configurations \(\{z_1\}\), \(\{z_2\}\), and \(\{z_3\}\). Suppose that \(\{z_2\}\) results in the lowest workload cost. The second step tries to expand \(\{z_2\}\) by adding one more index, which leads to two candidate configurations \(\{z_1, z_2\}\) and \(\{z_2, z_3\}\). Suppose that \(\{z_1, z_2\}\) is better and therefore returned by vanilla greedy. Note that the configuration \(\{z_1, z_3\}\) is never visited in this example. Vanilla greedy adopts a simple “first come first serve (FCFS)” budget allocation policy to make what-if calls.

Two-phase greedy. Figure 3(b) illustrates the two-phase greedy algorithm that can be viewed as an optimization on top of vanilla greedy. Specifically, there are two phases of greedy search in two-phase greedy. In the first phase, we view each query as a workload by itself and run vanilla greedy on top of it to obtain the best configuration for that query. In this particular example, we have three queries \(q_1\), \(q_2\), and \(q_3\) in the workload. After running vanilla greedy, we obtain their best configurations \(C_1^*, C_2^*,\) and \(C_3^*\), respectively. In the second phase, we take the union of the best configurations found for individual queries and use that as the refined set of candidate indexes for the entire workload. We then run vanilla greedy again for the workload with this refined set of candidate indexes, as depicted in Figure 3(b) for the given example. Two-phase greedy has particular importance in practice as it has been adopted by commercial index tuning software such as Microsoft’s Database Tuning Advisor (DTA) [1, 7]. Again, budget is allocated with the simple FCFS policy—the same as in vanilla greedy.

MCTS. Figure 4 illustrates the MCTS algorithm with the same example used in Figure 3. It is an iterative procedure that allocates one what-if call in each iteration until the budget runs out. The decision procedure in each iteration on which query and which configuration to issue the what-if call is an application of the classic Monte Carlo tree search (MCTS) algorithm [3] in the context of index configuration search. It involves four basic steps: (1) selection, (2) expansion, (3) simulation, and (4) update. Due to space limitation, we refer the readers to [46] for the full details of this procedure. After all what-if calls are issued, we run vanilla greedy again without making extra what-if calls to find the best configuration. Our particular version of MCTS here employs an \(\epsilon\)-greedy policy [39] when selecting the next index to explore.

3 WHAT-IF CALL INTERCEPTION

We develop “Wii” that can skip spurious what-if calls where their what-if costs and derived costs are close. One key idea is to develop a lower bound for the what-if cost: if the gap between the
lower bound and the derived cost is small, then it is safe to skip the what-if call. In this section, we present the generic form of the lower bound, as well as a confidence-based framework used by Wii on top of the lower bound to skip spurious what-if calls. We defer the discussion on further optimizations of the lower bound to Section 4.

3.1 Lower Bound of What-if Cost
We use $L(q, C)$ to denote the lower bound of the what-if cost $c(q, C)$. In the following, we first introduce the notion of marginal cost improvement (MCI) of an index, which indicates the additional benefit of adding this index to a configuration for a query. We then establish $L(q, C)$ by leveraging the upper bounds of MCI.

**Definition 2 (Marginal Cost Improvement).** We define the marginal cost improvement (MCI) of an index $z$ with respect to a query $q$ and a configuration $X$ as

$$
\delta(q, z, X) = c(q, X) - c(q, X \cup \{z\}).
$$

(2)

**Definition 3 (Cost Improvement).** We define the cost improvement (CI) of a query $q$ given a configuration $X$ as

$$
\Delta(q, X) = c(q, \emptyset) - c(q, X).
$$

(3)

We can express CI in terms of MCI. Specifically, consider a query $q$ and a configuration $C = \{z_1, ..., z_m\}$. The cost improvement $\Delta(q, C)$ can be seen as the sum of MCI’s by adding the indexes from $C$ one by one, namely,

$$
\Delta(q, C) = \left(c(q, \emptyset) - c(q, \{z_1\})\right) + \left(c(q, \{z_1\}) - c(q, \{z_1, z_2\})\right) + \cdots + \left(c(q, \{z_1, ..., z_{m-1}\}) - c(q, C)\right).
$$

Let $C_0 = \emptyset$ and $C_j = C_{j-1} \cup \{z_j\}$ for $1 \leq j \leq m$. It follows that $C_m = C$ and therefore, $\Delta(q, C) = \sum_{j=1}^{m} \delta(q, z_j, C_{j-1})$.

If we can have a configuration-independent upper bound $u(q, z_j)$ for $\delta(q, z_j, C_{j-1})$, namely, $u(q, z_j) \geq \delta(q, z_j, X)$ for any $X$, then

$$
\Delta(q, C) \leq \sum_{j=1}^{m} u(q, z_j).
$$

As a result,

$$
c(q, \emptyset) - c(q, C) \leq \sum_{j=1}^{m} u(q, z_j),
$$

and it follows that

$$
c(q, C) \geq c(q, \emptyset) - \sum_{j=1}^{m} u(q, z_j).
$$

We therefore can set the lower bound $L(q, C)$ as

$$
L(q, C) = c(q, \emptyset) - \sum_{j=1}^{m} u(q, z_j).
$$

(4)

**Generalization.** This idea can be further generalized if we know the what-if costs of configurations that are subsets of $C$. Specifically, let $S \subseteq C$ be a subset of $C$ with known what-if cost $c(q, S)$. Without loss of generality, let $C - S = \{z_1, ..., z_k\}$. We have

$$
c(q, S) - c(q, C) = \sum_{i=1}^{k} \left(c(q, C_{i-1}) - c(q, C_i)\right) \leq \sum_{i=1}^{k} u(q, z_i),
$$

where $C_0$ is now set to $S$. Therefore,

$$
c(q, C) \geq c(q, S) - \sum_{i=1}^{k} u(q, z_i).
$$
Since $S$ is arbitrary, we conclude

$$c(q, C) \geq \max_{S \subset C} \left( c(q, S) - \sum_{z \in C - S} u(q, z) \right).$$

As a result, it is safe to set

$$L(q, C) = \max_{S \subset C} \left( c(q, S) - \sum_{z \in C - S} u(q, z) \right). \tag{5}$$

Since $\emptyset \subset C$, Equation 5 is a generalization of Equation 4.

### 3.2 Upper Bound of MCI

The main question is then to maintain an upper bound $u(q, z)$ for the MCI of each query $q$ and each individual index $z$ so that $u(q, z) \geq \delta(q, z, X)$ for any configuration $X$. Below we discuss several such upper bounds. Our basic idea is to leverage the CIs of explored configurations that contain $z$, along with some well-known properties, such as monotonicity and submodularity, of the cost function used by the query optimizer.

#### 3.2.1 Naive Upper Bound

Let $\Omega$ be the set of all candidate indexes.

**Definition 4 (Naive Upper Bound).** Under Assumption 1,

$$u(q, z) = c(q, \emptyset) - c(q, \Omega) = \Delta(q, \Omega) \tag{6}$$

is a valid upper bound of $\delta(q, z, X)$ for any $X$.

Intuitively, by the monotonicity property, the MCI of any single index $z$ cannot be larger than the CI of all candidate indexes in $\Omega$ combined. In practical index tuning applications, we often have $c(q, \Omega)$ available. However, if $c(q, \Omega)$ is unavailable, then we set $u(q, z) = c(q, \emptyset)$ as it always holds that $c(q, \Omega) \geq 0$.

#### 3.2.2 Upper Bound by Submodularity

We can improve over the naive upper bound by assuming that the cost function is submodular, which has been studied by previous work [10].

**Assumption 2 (Submodularity).** Given two configurations $X \subseteq Y$ and an index $z \notin Y$, we have

$$c(q, Y) - c(q, Y \cup \{z\}) \leq c(q, X) - c(q, X \cup \{z\}). \tag{7}$$

Or equivalently, $\delta(q, z, Y) \leq \delta(q, z, X)$.

That is, the MCI of an index $z$ diminishes when $z$ is included into larger configuration with more indexes. Submodularity does not hold often due to index interaction [31]. We also validated the submodularity assumption using Microsoft SQL Server and the same workloads that we used to validate the monotonicity assumption. Our validation results show that submodularity holds with probability between 0.75 and 0.89 on the workloads tested [41].

**Lemma 1.** Under Assumption 2, we have

$$\delta(q, z, X) \leq \Delta(q, \{z\})$$

for any configuration $X$.

Due to space constraint, all proofs are postponed to the full version of this paper [41]. Intuitively, Lemma 1 indicates that the CI of a singleton configuration $\{z\}$ can be used as an upper bound of the MCI of the index $z$. As a result, we can set

$$u(q, z) = \Delta(q, \{z\}) = c(q, \emptyset) - c(q, \{z\}). \tag{8}$$
There are cases where \( c(q, \{z\}) \) is unknown but we know the cost of some configuration \( X \) that contains \( z \), e.g., in MCTS where configurations are explored in random order. By Assumption 1, 
\[
c(q, \{z\}) \geq \max_{z \in X} c(q, X).
\]
Therefore, we can generalize Equation 8 to have

**Definition 5 (Submodular Upper Bound).**
\[
u(q, z) = c(q, \emptyset) - \max_{z \in X} c(q, X)
\]
\[
= \min_{z \in X} \left( c(q, \emptyset) - c(q, X) \right)
\]
\[
= \min_{z \in X} \Delta(q, X).
\]
That is, the MCI of an index should be no larger than the minimum CI of all the configurations that contain it.

3.2.3 **Summary.** To summarize, assuming monotonicity and submodularity of the cost function \( c \), we can set \( u(q, z) \) as follows:

\[
u(q, z) = \min\{c(q, \emptyset), \Delta(q, \Omega), \Delta(q, \{z\}), \min_{z \in X} \Delta(q, X)\}.
\]

3.3 **Confidence-based What-if Call Skipping**

Intuitively, the confidence of skipping the what-if call for a QCP \((q, C)\) depends on the closeness between the lower bound \( L(q, C) \) and the upper bound \( U(q, C) \), i.e., the derived cost \( d(q, C) \). We define the gap between \( U(q, C) \) and \( L(q, C) \) as
\[
G(q, C) = U(q, C) - L(q, C).
\]
Clearly, the larger the gap is, the lower the confidence is. Therefore, it is natural to define the confidence as
\[
\alpha(q, C) = 1 - \frac{G(q, C)}{U(q, C)} = \frac{L(q, C)}{U(q, C)}.
\]
Following this definition, we have \( 0 \leq \alpha(q, C) \leq 1 \). We further note two special cases: (1) \( \alpha(q, C) = 0 \), which implies \( L(q, C) = 0 \); and (2) \( \alpha(q, C) = 1 \), which implies \( L(q, C) = U(q, C) \).

Let \( \alpha \in [0, 1] \) be a threshold for the confidence, i.e., it is the minimum confidence for skipping a what-if call and we require \( \alpha(q, C) \geq \alpha \). Intuitively, the higher \( \alpha \) is, the higher confidence that a what-if call can be skipped with. In our experimental evaluation, we further varied \( \alpha \) to test the effectiveness of this confidence-based interception mechanism (see Section 6).

4 **OPTIMIZATION**

We present two optimization techniques for the generic lower bound detailed in Section 3.1, which is agnostic to budget-aware configuration enumeration algorithms—it only relies on general assumptions (i.e., monotonicity and submodularity) of the cost function \( c \). One optimization is dedicated to budget-aware greedy search (i.e., *vanilla/two-phase greedy*), which is of practical importance due to its adoption in commercial index tuning software [7] (Section 4.1). The other optimization is more general and can also be used for other configuration enumeration algorithms mentioned in Section 2.2.3 such as MCTS (Section 4.2).
4.1 MCI Upper Bounds for Greedy Search

We propose the following optimization procedure for maintaining the MCI upper-bound $u(q, z)$, which is the basic building block of the lower bound presented in Section 3.1, in vanilla greedy and two-phase greedy (see Section 2):

**Procedure 1.** For each index $z$ that has not been selected by greedy search, we can update $u(q, z)$ w.r.t. the current configuration selected by greedy search as follows:

1. **Initialize** $u(q, z) = \min \{c(q, \emptyset), \Delta(q, \Omega)\}$ for each index $z$.
2. **During each greedy step** $1 \leq k \leq K$, update:
   $$u(q, z) = c(q, C_{k-1}) - c(q, C_{k-1} \cup \{z\}) = \delta(q, z, C_{k-1})$$
   
   if both $c(q, C_{k-1})$ and $c(q, C_{k-1} \cup \{z\})$ are available.

In step (2), $C_k$ is the configuration selected by greedy search in step $k$ and we set $C_0 = \emptyset$. A special case is when $k = 1$, if we know $c(q, \{z\})$ then we can update $u(q, z) = c(q, \emptyset) - c(q, \{z\}) = \Delta(q, \{z\})$, which reduces to the general upper bound (see Lemma 1).

**Theorem 1.** Under Assumptions 1 and 2, Procedure 1 is correct, i.e., the $u(q, z)$ after each update remains an MCI upper bound w.r.t. any future configuration $X$ explored by greedy search.

4.2 Coverage-based Refinement

The tightness of the MCI upper bounds in Section 3.2 largely depends on the knowledge about $c(q, \{z\})$, namely, what-if costs of singleton configurations with one single index. Unfortunately, such information is often unavailable, and the MCI upper bound in Equation 9 is reduced to its naive version (Equation 6). For vanilla greedy and two-phase greedy, this implies that none of the QCP’s with singleton configurations can be skipped under a reasonable confidence threshold (e.g., 0.8), which can take a large fraction of the budget, although the bounds are effective at skipping what-if calls for multi-index configurations; for MCTS where configurations are explored in a random order, this further implies that skipping can be less effective for multi-index configurations as they are more likely to contain indexes with unknown what-if costs, in contrast to greedy search where multi-index configurations are always explored after singleton configurations. To overcome this limitation, we propose refinement techniques based on estimating the what-if cost $c(q, \{z\})$ if it is unknown, by introducing the notion of “coverage.”

4.2.1 Definition of Coverage. We assume that $c(q, \Omega)$ is known for each query $q$. Moreover, we assume that we know the subset $\Omega_q \subset \Omega$ of indexes that appear in the optimal plan of $q$ by using indexes in $\Omega$. Clearly, $c(q, \Omega) = c(q, \Omega_q)$.

For an index $z$, we define its coverage on the query $q$ as

$$\rho(q, z) = \frac{c(q, \emptyset) - c(q, \{z\})}{c(q, \emptyset) - c(q, \Omega_q)} = \frac{\Delta(q, \{z\})}{\Delta(q, \Omega_q)}.$$  \hfill (11)

In other words, coverage measures the relative cost improvement of $z$ w.r.t. the maximum possible cost improvement over $q$ delivered by $\Omega_q$. If we know $\rho(q, z)$, the cost $c(q, \{z\})$ can be recovered as

$$c(q, \{z\}) = c(q, \emptyset) - \rho(q, z) \cdot \left( c(q, \emptyset) - c(q, \Omega_q) \right)$$

$$= \left( 1 - \rho(q, z) \right) \cdot c(q, \emptyset) + \rho(q, z) \cdot c(q, \Omega_q).$$

In the following, we present techniques to estimate $\rho(q, z)$ based on the similarities between index configurations, in particular $\{z\}$ and $\Omega_q$. 

4.2.2 Estimation of Coverage. We estimate coverage based on the assumption that it depends on the similarity between \( \{z\} \) and \( \Omega_q \). Specifically, let \( \text{Sim}(\{z\}, \Omega_q) \) be some similarity measure that is between 0 and 1, and we define

\[
\rho(q, z) = \text{Sim}(\{z\}, \Omega_q).
\]

The problem is then reduced to developing an appropriate similarity measure. Our current solution is the following, while further improvement is possible and left for future work.

Configuration Representation. We use a representation similar to the one described in DBA bandits [28] that converts an index \( z \) into a feature vector \( \tilde{z} \). Specifically, we use one-hot encoding based on all indexable columns identified in the given workload \( W \). Let \( D = \{c_1, ..., c_L\} \) be the entire domain of these \( L \) indexable columns. For a given index \( z \), \( \tilde{z} \) is an \( L \)-dimensional vector. If some column \( c_l \in D \) \( (1 \leq l \leq L) \) appears in \( z \), then \( \tilde{z}[l] \) receives some nonzero weight \( w_l \) based on the weighing policy described below:

- If \( c_l \) is the \( j \)-th key column of \( z \), \( w_l = \frac{1}{J} \); 
- If \( c_l \) is an included column of \( z \), \( w_l = \frac{1}{J} \) where \( J \) is the number of key columns contained by \( z \).

Otherwise, we set \( \tilde{z}[l] = 0 \). Note that the above weighing policy considers the columns contained by an index as well as their order. Intuitively, leading columns in index keys play a more important role than other columns (e.g., for a “range predicate”, an access path chosen by the query optimizer needs to match the “sort order” specified in the index key columns).

We further combine feature vectors of individual indexes to generate a feature vector for the entire configuration. Specifically, consider a configuration \( C = \{z_1, ..., z_m\} \) and let \( \tilde{z}_i \) be the feature representation of the index \( z_i \) \( (1 \leq i \leq m) \). The feature representation \( \tilde{C} \) of \( C \) is again an \( L \)-dimensional vector where

\[
\tilde{C}[l] = \max(\{\tilde{z}_1[l], ..., \tilde{z}_m[l]\}), \text{ for } 1 \leq l \leq L.
\]

That is, the weight \( \tilde{C}[l] \) is the largest weight of the \( l \)-th dimension among the indexes contained by \( C \). In particular, we generate the feature vector \( \Omega_q \) for \( \Omega_q \) in this way.

Query Representation. We further use a representation similar to the one described in ISUM [35] to represent a query \( q \) as a feature vector \( \tilde{q} \). Specifically, we again use one-hot encoding for the query \( q \) with the same domain \( D = \{c_1, ..., c_L\} \) of all indexable columns. If some column \( c_l \in D \) appears in the query \( q \), we assign a nonzero weight to \( \tilde{q}[l] \); otherwise, \( \tilde{q}[l] = 0 \). Here, we use the same weighing mechanism as used by ISUM. That is, the weight of a column is computed based on its corresponding table size and the number of candidate indexes that contain it. The intuition is that a column from a larger table and contained by more candidate indexes is more important and thus is assigned a higher weight.

Similarity Measure. Before measuring the similarity, we first project \( \tilde{z} \) and \( \Omega_q \) onto \( \tilde{q} \) to get their images under the context of the query \( q \). The projection is done by taking the element-wise dot product, i.e., \( \tilde{z} \cdot \tilde{q} \) and \( \Omega_q \cdot \tilde{q} \). Note that \( \tilde{z} \) and \( \Omega_q \) remain vectors. We now define the similarity measure as

\[
\text{Sim}(\{z\}, \Omega_q) = \frac{\langle \tilde{z}, \Omega_q \rangle}{|\Omega_q|^2} = \frac{|\tilde{z}| \cdot |\Omega_q| \cdot \cos \theta}{|\Omega_q|^2} = \frac{|\tilde{z}| \cdot \cos \theta}{|\Omega_q|}, \tag{12}
\]

where \( \theta \) represents the angle between the two vectors \( \tilde{z} \) and \( \Omega_q \).

Figure 5 illustrates and contrasts the definition and estimation of coverage. Figure 5(a) highlights the observation that \( c(q, \{z\}) \) must lie between \( c(q, \Omega_q) \) and \( c(q, \emptyset) \), and coverage measures the cost improvement \( \Delta(q, \Omega_q) \) of \( \Omega_q \) (i.e., the green segment) that is covered by the cost improvement \( \Delta(q, \{z\}) \) of \( \{z\} \) (i.e., the orange segment). On the other hand, Figure 5(b) depicts the geometric
\[ \rho = \frac{c(q, \emptyset) - c(q, \{z\})}{c(q, \emptyset) - c(q, \Omega_q)} \]

5 INTEGRATION

In this section, we present design considerations and implementation details when integrating Wii with existing budget-aware configuration search algorithms. We start by presenting the API functions provided by Wii. We then illustrate how existing budget-aware configuration enumeration algorithms can leverage the Wii API’s without modification to the algorithms.

5.1 Wii API Functions

As illustrated in Figure 1, Wii sits between the index tuner and the query optimizer. It offers two API functions that can be invoked by a budget-aware configuration enumeration algorithm: (1) \texttt{InitMCIBounds} that initializes the MCI upper-bounds \( u(q, z) \); and (2) \texttt{EvalCost} that obtains the cost of a QCP \((q, C)\) in a budget-aware manner by utilizing the lower bound \( L(q, C) \) and the upper bound \( U(q, C) \), i.e., the derived cost \( d(q, C) \).

5.1.1 The \texttt{InitMCIBounds} Function. Algorithm 1 presents the details. It initializes the MCI upper bound \( u(q, z) \) for each query \( q \in W \) and each of its candidate indexes \( z \in I_q \). If \( c(q, \Omega_q) \) is available, it uses the naive upper bound (Equation 6); otherwise, it uses \( c(q, \emptyset) \).

5.1.2 The \texttt{EvalCost} Function. Algorithm 2 presents the details. If the what-if cost \( c(q, C) \) is known, it simply uses that and updates the MCI upper-bounds (lines 1 to 3). Otherwise, it checks whether the budget \( B \) on the number of what-if calls has been reached and returns the derived cost \( d(q, C) \) if so (lines 4 to 5). On the other hand, if there is remaining budget, i.e., \( B > 0 \), it then tries to use the upper-bound \( U(q, C) \) and the lower-bound \( L(q, C) \) to see whether the what-if call for \((q, C)\) can be...
skipped; if so, the derived cost \( d(q, C) \) is returned (lines 6 to 11)—the budget \( B \) remains the same in this case. Finally, if the confidence of skipping is low, we make one what-if call to obtain \( c(q, C) \) (lines 12 to 13) and update the MCI upper-bounds (line 14). As a result, we deduct one from the current budget \( B \) (line 15).

One may have noticed the optional input parameter \( S \) in Algorithm 2, which represents some subset configuration of \( C \) and is set to be the existing configuration \( \emptyset \) by default. We will discuss how to specify this parameter when using Wii in existing budget-aware configuration enumeration algorithms (e.g., greedy search and MCTS) shortly.

### 5.2 Budget-aware Greedy Search

To demonstrate how to use the Wii API's without modifying the existing budget-aware configuration search algorithms, Algorithm 3 showcases how these API's can be used by budget-aware greedy search, a basic building block of the existing algorithms. Notice that the \texttt{InitMCIBounds} API is invoked at line 1, whereas the \texttt{EvalCost} API is invoked at line 9, which are the only two differences compared to regular budget-aware greedy search. Therefore, there is no \textit{intrusive} change to the greedy search procedure itself.

**Remarks.** We have two remarks here. First, when calling Wii to evaluate cost at line 9, we pass \( C^* \) to the optional parameter \( S \) in Algorithm 2. Note that this is just a special case of Equation 5 for greedy search, as stated by the following theorem:

\begin{algorithm}
\textbf{Algorithm 2: EvalCost}(q, C, B, \alpha, S \leftarrow \emptyset)
\begin{algorithmic}[1]
\State \textbf{Input:} \( q \), the query; \( C \), the configuration; \( B \), the budget on the number of what-if calls; \( \alpha \), the threshold on the confidence \( \alpha(q, C) \); \( S \), an (optional) subset of \( C \) with known what-if cost \( c(q, S) \), which defaults to the existing configuration \( \emptyset \).
\State \textbf{Output:} \( \text{cost}(q, C) \), the cost of \( q \) w.r.t. \( C \); \( B' \), the remaining budget.
\If {\( c(q, C) \) \text{ is known}}
\State UpdateMCIBounds\texttt{(C, S)};
\State \Return \( (\text{cost}(q, C) \leftarrow c(q, C), B' \leftarrow B) \);
\EndIf
\If {\( B \) \text{ is zero}}
\State \Return \( (\text{cost}(q, C) \leftarrow d(q, C), B' \leftarrow 0) \);
\EndIf
\State \( U(q, C) \leftarrow d(q, C) \);
\State \( L(q, C) \leftarrow \max\{0, c(q, \Omega_q), c(q, S) - \sum_{x \in C \setminus S} u(q, x)\} \);
\If {\( \alpha(q, C) = \frac{L(q, C)}{U(q, C)} \geq \alpha \)}
\State \Return \( (\text{cost}(q, C) \leftarrow d(q, C), B' \leftarrow B) \);
\EndIf
\State \( \text{Need to go to the query optimizer to get } c(q, C) \).
\State \( c(q, C) \leftarrow \text{WhatIfCall}(q, C) \);
\State UpdateMCIBounds\texttt{(C, S)};
\State \Return \( (\text{cost}(q, C) \leftarrow c(q, C), B' \leftarrow B - 1) \);
\EndIf
\State UpdateMCIBounds\texttt{(C, S)};
\State \texttt{foreach} \( x \in C \setminus S \) \texttt{do}
\State \( u(q, x) \leftarrow \min\{u(q, x), c(q, S) - c(q, C)\} \);
\end{algorithmic}
\end{algorithm}

Theorem 2. In the context of greedy search, Equation 5 reduces to
\[ L(q, C_z) = c(q, C^*) - \sum_{x \in C_z - C^*} u(q, x) = c(q, C) - u(q, z), \]
where \( C_z = C^* \cup \{z\} \) and \( C^* \) is the latest configuration selected by budget-aware greedy search (as shown in Algorithm 3).

Second, in the context of greedy search, the update step at line 20 of Algorithm 2 becomes
\[ u(q, x) \leftarrow \min \{u(q, x), c(q, C^*) - c(q, C)\}. \]
The correctness of this update has been given by Theorem 1.

Algorithm 3: GreedySearch\((W, I, K, B, \alpha)\)
\[
\text{InitMCIBounds}(W, I); \\
C^* \leftarrow \emptyset, \text{cost}^* \leftarrow \text{cost}(W, \emptyset), B' \leftarrow B; \\
\text{while } I \neq \emptyset \text{ and } |C^*| < K \text{ do} \\
\quad C \leftarrow C^*, \text{cost} \leftarrow \text{cost}^*; \\
\quad \text{foreach index } z \in I \text{ do} \\
\quad\quad C_z \leftarrow C^* \cup \{z\}; \\
\quad\quad \text{cost}(W, C_z) \leftarrow 0; \\
\quad\quad \text{foreach } q \in W \text{ do} \\
\quad\quad\quad \left(\text{cost}(q, C_z), B'\right) \leftarrow \text{EvalCost}(q, C_z, B', \alpha, C^*); \\
\quad\quad\quad \text{cost}(W, C_z) \leftarrow \text{cost}(W, C_z) + \text{cost}(q, C_z); \\
\quad\quad\quad \text{if } \text{cost}(W, C_z) < \text{cost} \text{ then} \\
\quad\quad\quad\quad C \leftarrow C_z, \text{cost} \leftarrow \text{cost}(W, C_z); \\
\quad\quad\quad\quad \text{if } \text{cost} \geq \text{cost}^* \text{ then} \\
\quad\quad\quad\quad\quad \text{return } (C^*, B'); \\
\quad\quad\quad\quad \text{else} \\
\quad\quad\quad\quad\quad C^* \leftarrow C, \text{cost}^* \leftarrow \text{cost}, I \leftarrow I - C^*; \\
\quad\text{return } (C^*, B'); 
\]

5.3 Budget-aware Configuration Enumeration

We now outline the skeleton of existing budget-aware configuration enumeration algorithms after integrating Wii. We use the integrated budget-aware greedy search procedure in Algorithm 3 as a building block in our illustration.

5.3.1 Vanilla Greedy. The vanilla greedy algorithm after integrating Wii is exactly the same as the GreedySearch procedure presented by Algorithm 3.

5.3.2 Two-phase Greedy. Algorithm 4 presents the details of the two-phase greedy algorithm after integrating Wii. There is no change to two-phase greedy except for using the version of GreedySearch in Algorithm 3. The function GetCandidateIndexes selects a subset of candidate indexes \( I_q \) from \( I \), considering only the indexable columns contained by the query \( q \) [8].
Algorithm 4: TwoPhaseGreedy($W, I, K, B, \alpha$)

**Input:** $W$, the workload; $I$, the candidate indexes; $K$, the cardinality constraint; $B$, the budget on the number of what-if calls; $\alpha$, the confidence threshold.

**Output:** $C^*$, the best configuration; $B'$, the remaining budget.

1. $I_W \leftarrow \emptyset$, $B' \leftarrow B$;
2. foreach $q \in W$ do
   3. $I_q \leftarrow \text{GetCandidateIndexes}(q, I)$;
   4. $(C_q, B') \leftarrow \text{GreedySearch} \{q\}, I_q, K, B', \alpha$;
   5. $I_W \leftarrow I_W \cup C_q$;
   6. $(C^*, B') \leftarrow \text{GreedySearch}(W, I_W, K, B', \alpha)$;
3. return $(C^*, B')$;

5.3.3 MCTS. Algorithm 5 presents the skeleton of MCTS after Wii is integrated. The details of the three functions InitMCTS, SelectListQueryConfigByMCTS, and UpdateRewardForMCTS can be found in [46]. Again, there is no change to the MCTS algorithm except for that cost evaluation at line 5 is delegated to the EvalCost API of Wii (Algorithm 2).

Note that here we pass the existing configuration $\emptyset$ to the optional parameter $S$ in Algorithm 2, which makes line 8 of Algorithm 2 on computing $L(q, C)$ become

$$L(q, C) \leftarrow \max \{0, c(q, \Omega_q), c(q, \emptyset) - \sum_{x \in C} u(q, x)\}.$$  

Essentially, this means that we use Equation 4 for $L(q, C)$, instead of its generalized version shown in Equation 5. Although we could have used Equation 5, it was our design decision to stay with Equation 4, not only for simplicity but also because of the inefficacy of Equation 5 in the context of MCTS. This is due to the fact that in MCTS configurations and queries are explored in random order. Therefore, the subsets $S$ w.r.t. a given pair of $q$ and $C$ with known what-if costs $c(q, S)$ are sparse. As a result, Equation 5 often reduces to Equation 4 when running Wii underlying MCTS.

Algorithm 5: MCTS($W, I, K, B, \tau$)

**Input:** $W$, the workload; $I$, the candidate indexes; $K$, the cardinality constraint; $B$, the budget on the number of what-if calls; $\alpha$, the confidence threshold.

**Output:** $C^*$, the best configuration; $B'$, the remaining budget.

1. $B' \leftarrow B$;
2. InitMCTS($W, I$);
3. while $B' > 0$ do
   4. $(q, C) \leftarrow \text{SelectQueryConfigByMCTS}(W, I, K)$;
   5. $(\text{cost}(q, C), B') \leftarrow \text{EvalCost}(q, C, B', \alpha, \emptyset)$;
   6. UpdateRewardForMCTS($q, C, \text{cost}(q, C)$);
   7. $(C^*, B') \leftarrow \text{GreedySearch}(W, I, K, B', \alpha)$;
4. return $(C^*, B')$;

6 EXPERIMENTAL EVALUATION

We now report experimental results on evaluating Wii when integrated with existing budget-aware configuration search algorithms. We perform all experiments using Microsoft SQL Server 2017 under Windows Server 2022, running on a workstation equipped with 2.6 GHz multi-core AMD CPUs and 256 GB main memory.

6.1 Experiment Settings

Datasets. We used standard benchmarks and real workloads in our study. Table 1 summarizes the information of the workloads. For benchmark workloads, we use both the TPC-H and TPC-DS benchmarks with scaling factor 10. We also use two real workloads, denoted 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. Moreover, we report the number of candidate indexes of each workload, which serves as an indicator of the size of the corresponding search space faced by an index configuration search algorithm.

Algorithms Evaluated. We focus on two state-of-the-art budget-aware configuration search algorithms described in Section 2: (1) two-phase greedy, which has been adopted by commercial index tuning software [7]; and (2) MCTS, which shows better performance than two-phase greedy. We omit vanilla greedy as it is significantly inferior to two-phase greedy [46]. Both two-phase greedy and MCTS use derived cost as an estimate for the what-if cost when the budget on what-if calls is exhausted. We evaluate Wii when integrated with the above configuration search algorithms.

Other Experimental Settings. In our experiments, we set the cardinality constraint $K \in \{10, 20\}$. Since the TPC-H workload is relatively small compared to the other workloads, we varied the budget $B$ on the number of what-if calls in $\{500, 1000\}$; for the other workloads, we varied the budget $B$ in $\{500, 1000, 2000, 5000\}$.

### Table 1. Summary of database and workload statistics.

<table>
<thead>
<tr>
<th>Name</th>
<th>DB Size</th>
<th># Queries</th>
<th># Tables</th>
<th>Avg. # Joins</th>
<th>Avg. # Scans</th>
<th># Candidate Indexes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H</td>
<td>sf=10</td>
<td>22</td>
<td>8</td>
<td>2.8</td>
<td>3.7</td>
<td>168</td>
</tr>
<tr>
<td>TPC-DS</td>
<td>sf=10</td>
<td>99</td>
<td>24</td>
<td>7.7</td>
<td>8.8</td>
<td>848</td>
</tr>
<tr>
<td>Real-D</td>
<td>587GB</td>
<td>32</td>
<td>7,912</td>
<td>15.6</td>
<td>17</td>
<td>417</td>
</tr>
<tr>
<td>Real-M</td>
<td>26GB</td>
<td>317</td>
<td>474</td>
<td>20.2</td>
<td>21.7</td>
<td>4,490</td>
</tr>
</tbody>
</table>

### 6.2 End-to-End Improvement

The evaluation metric used in our experiments is the percentage improvement of the workload based on the final index configuration found by a search algorithm, defined as

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

where $c(W, C) = \sum_{q \in W} c(q, C)$. Note that here we use the query optimizer’s what-if cost estimate $c(q, C)$ as the gold standard of query execution cost, instead of using the actual query execution time, to be in line with previous work on evaluating index configuration enumeration algorithms [8, 19].

#### 6.2.1 Two-phase Greedy

Figure 6 presents the evaluation results of Wii for two-phase greedy when setting the confidence threshold $\alpha = 0.9$ (see Section 6.2.5 for details of the ‘Best’ lines). We observe that Wii significantly outperforms the baseline (i.e., two-phase greedy without what-if call interception). For example, when setting $K = 20$ and $B = 5,000$, Wii improves over the baseline by increasing the percentage improvement from 50% to 65% on TPC-DS (Figure 6(f)), from 58% to 74% on Real-D (Figure 6(g)), and from 32% to 54% on Real-M (Figure 6(h)); even for the smallest workload TPC-H, when setting $K = 20$ and $B = 1,000$, Wii improves over the baseline from 78% to 86% (Figure 6(e)). Note that here Wii has used the optimization for greedy search (Section 4.1).

We also observe that incorporating the coverage-based refinement described in Section 4.2 can further improve Wii in certain cases. For instance, on TPC-DS when setting $K = 20$ and $B = 2,000$, it improves Wii by 13%, i.e., from 49% to 62%, whereas Wii and the baseline perform similarly (Figure 6(f)); on Real-D when setting $K = 10$ and $B = 500$ (Figure 6(c)), it improves Wii by an
Fig. 6. Results for *two-phase greedy* with confidence threshold $\alpha = 0.9$ (“Cov.” is shorthand for “Coverage”).

Fig. 7. Impact on the performance of Wii with or without the optimization for the MCI upper bounds ($\alpha = 0.9$).

Fig. 8. Results for *MCTS* with confidence threshold $\alpha = 0.9$ (“Cov.” is shorthand for “Coverage”).

additional percentage improvement of 17.8% (i.e., from 45.3% to 63.1%), which translates to 32.2% improvement over the baseline (i.e., from 30.9% to 63.1%).

**Impact of Optimization for MCI Upper Bounds.** We further study the impact of the optimization proposed in Section 4.1 for *two-phase greedy*. In our experiment, we set $\alpha = 0.9$, $B = 1,000$ for **TPC-H** and $B = 5,000$ for the other workloads. Figure 7 presents the results. We observe that the optimization for MCI upper bounds offers a differentiable benefit in *two-phase greedy* on **TPC-H**, **TPC-DS**, and **Real-M**. Given its negligible computation overhead, this optimization is warranted to be enabled by default in Wii.

6.2.2 MCTS. Figure 8 presents the results of Wii for MCTS, again by setting the confidence threshold $\alpha = 0.9$. Unlike the case of *two-phase greedy*, for MCTS Wii often performs similarly...
6.2.3 Discussion. Comparing Figures 6 and 8, while the baseline version of two-phase greedy clearly underperforms that of MCTS, the Wii-enhanced version of two-phase greedy performs similarly or even better than that of MCTS. Existing budget allocation policies are largely macro-level optimization mechanisms, meaning that they deem what-if calls as atomic black-box operations that are out of their optimization scopes. However, our results here reveal that micro-level optimization mechanisms like Wii that operate at the granularity of individual what-if calls can interact with and have profound impact on the performance of those macro-level optimization mechanisms. An in-depth study and understanding of such macro-/micro-level interactions may lead to invention of better budget allocation policies.

Moreover, based on our evaluation results, the coverage-based refinement does not always improve Wii’s performance. A natural question is then how users would choose whether or not to use it. Are there some simple tests that can indicate whether or not it will be beneficial? Since the motivation of the coverage-based refinement is to make Wii work more effectively in the presence of unknown singleton-configuration what-if costs, one idea could be to measure the fraction of such singleton-configurations and enable the coverage-based refinement only when this fraction is high. However, this measurement can only be monitored “during” index tuning and there are further questions if index tuning is budget-constrained (e.g., how much budget should be allocated for monitoring this measurement). Thus, there seems to be no simple answer and we leave its investigation for future work.

6.2.4 Evaluation of Confidence-based What-if Call Skipping. We start by investigating the impact of the confidence threshold $\alpha$ on Wii. For this set of experiments, we use the budget $B = 1,000$ for TPC-H and use $B = 5,000$ for the other workloads, and we vary $\alpha \in \{0.8, 0.9, 0.95\}$. Figures 10 and 11 present the evaluation results. We observe that Wii is not sensitive to the threshold $\alpha$
within the range that we tested, for both two-phase greedy and MCTS. On the other hand, coverage-based refinement is more sensitive to \( \alpha \). For instance, for two-phase greedy on Real-M with cardinality constraint \( K = 10 \) (ref. Figure 10(d)), the end-to-end percentage improvement of the final configuration found increases from 35.6% to 53.3% when raising \( \alpha \) from 0.8 to 0.95. This suggests both opportunities and risks of using the coverage-based refinement for Wii, as one needs to choose the confidence threshold \( \alpha \) more carefully. A more formal analysis can be found in [41].

**Low Confidence Threshold.** An interesting question is the performance impact of using a relatively lower confidence threshold compared to the ones used in the previous evaluations. To investigate this question, we further conduct experiments by setting the confidence threshold \( \alpha = 0.5 \). Figures 9 and 12 present results for two-phase greedy and MCTS with the cardinality constraint \( K = 20 \). We have the following observations. First, the performance of Wii often becomes much worse compared to using a high confidence threshold like the \( \alpha = 0.9 \) in the charts—it is sometimes even worse than the baseline, e.g., in the case of MCTS on Real-D, as shown in Figure 12(c). Second, coverage-based refinement seems more sensitive to the use of a low confidence threshold, due to its inherent uncertainty of estimating singleton-configuration what-if costs.

**Necessity of Confidence-based Mechanism.** Since the confidence-based skipping mechanism comes with additional overhead of computing the lower and upper bounds of what-if cost (Section 6.4), it
MCTS
0.95
MCTS
0.95
MCTS
0.95
MCTS
0.95
Average Computation Time (ms)
Average Computation Time (ms)
Average Computation Time (ms)
Average Computation Time (ms)
15
30
40
12
30
10
35
40
16
18
45
1000
10000
0.2
0.2
0.5
0.6
0.8
0.8
1.2
0.3
1.2
1.4
1.2
1.8
0.01
0.01
2
5
2
4
6
8
100
0.1
0.1
0
0
0
0
2
4
6
8
10
12
14
16
18
0.8 0.9 0.95
What−If Calls Skipped (x1000)
Confidence α
Wii (K=10)
Cov. (K=10)
Wii (K=20)
Cov. (K=20)

6.3 Efficacy of What-If Call Interception

We measure the relative amount of what-if calls skipped by Wii, namely, the ratio between the number of what-if calls skipped and the budget allowed. Figures 14 and 15 present the results for two-phase greedy and MCTS when varying $\alpha \in \{0.8, 0.9, 0.95\}$.

We have several observations. First, in general, Wii is more effective at skipping spurious what-if calls for two-phase greedy than MCTS. For example, when setting $K = 20$ and $\alpha = 0.9$, Wii is able to skip 3.6B (i.e., $3.6 \times 5,000 = 18,000$) what-if calls for two-phase greedy whereas only 0.57B (i.e., 2,850) what-if calls for MCTS. This is correlated with the observation that Wii exhibits more significant end-to-end improvement in terms of the final index configuration found for two-phase greedy than MCTS, as we highlighted in Section 6.2. Second, the coverage-based refinement often enables Wii to skip more what-if calls. For instance, for MCTS on Real-M when setting $K = 20$ and $\alpha = 0.8$, Wii is able to skip only 1.48B (i.e., 7,400) what-if calls, which leads to no observable end-to-end

Fig. 14. Amount of what-if calls skipped by Wii for two-phase greedy (“Cov.” is shorthand for “Wii-Coverage”).

Fig. 15. Amount of what-if calls skipped by Wii for MCTS (“Cov.” is shorthand for “Wii-Coverage”).

Fig. 16. Average computation time (in milliseconds) of the lower bound on the what-if cost ($K = 20, \alpha = 0.9$). It is natural to ask whether such complexity is necessary. To justify this, we compare the confidence-based mechanism with a simple randomized mechanism that skips what-if calls randomly w.r.t. a given skipping probability threshold $p$. Figure 13 presents the results when setting $p = \alpha = 0.9$—we use the same confidence threshold for a fair comparison. We observe that the randomized mechanism performs similarly to the baseline but often much worse than Wii.

6.2.5 Best Possible Improvement.

It is difficult to know the best possible improvement without making a what-if call for every QCP enumerated during configuration search, which is infeasible in practice. We provide an approximate assessment by using a much larger budget $B$ in two-phase greedy. Specifically, we use $B = 5,000$ for TPC-H and $B = 20,000$ for the other workloads. For each workload, we run both two-phase greedy without and with Wii, and we take the best improvement observed in these two runs. The ‘Best’ line in Figures 6 and 8 presents this result.

6.3 Efficacy of What-If Call Interception

We measure the relative amount of what-if calls skipped by Wii, namely, the ratio between the number of what-if calls skipped and the budget allowed. Figures 14 and 15 present the results for two-phase greedy and MCTS when varying $\alpha \in \{0.8, 0.9, 0.95\}$.

We have several observations. First, in general, Wii is more effective at skipping spurious what-if calls for two-phase greedy than MCTS. For example, when setting $K = 20$ and $\alpha = 0.9$, Wii is able to skip 3.6B (i.e., $3.6 \times 5,000 = 18,000$) what-if calls for two-phase greedy whereas only 0.57B (i.e., 2,850) what-if calls for MCTS. This is correlated with the observation that Wii exhibits more significant end-to-end improvement in terms of the final index configuration found for two-phase greedy than MCTS, as we highlighted in Section 6.2. Second, the coverage-based refinement often enables Wii to skip more what-if calls. For instance, for MCTS on Real-M when setting $K = 20$ and $\alpha = 0.8$, Wii is able to skip only 1.48B (i.e., 7,400) what-if calls, which leads to no observable end-to-end
improvement over the baseline; with the coverage-based refinement enabled, however, the number of what-if calls that Wii can skip rises to 42.7B (i.e., 213,500), which results in nearly 10% boost on the end-to-end improvement (ref. Figure 11(d)). Third, while one would expect that the amount of what-if calls skipped decreases when we increase the confidence threshold $\alpha$, this is sometimes not the case, especially for two-phase greedy. As shown in Figures 14(a), 14(b), and 14(c), the number of skipped calls can actually increase when raising $\alpha$. The reason for this unexpected phenomenon is the special structure of the two-phase greedy algorithm: lowering $\alpha$ allows for more what-if calls to be skipped in the first phase where the goal is to find good candidate indexes for each individual query. Skipping more what-if calls in the first phase therefore can result in fewer candidate indexes being selected because, without what-if calls, the derived costs for the candidate indexes will have the same value (as the what-if cost with the existing index configuration, i.e., $c(q, \emptyset)$) and thus exit early in Algorithm 3 (line 14). As a result, it eventually leads to a smaller search space for the second phase and therefore fewer opportunities for what-if call interception.

### 6.4 Computation Overhead

We measure the average computation time of the lower bound of the what-if cost. For comparison, we also report the average time of cost derivation as well as making a what-if call. Figure 16 summarizes the results when running two-phase greedy and MCTS with $K = 20$ and $\alpha = 0.9$.

We have the following observations. First, the computation time of the lower bound is similar to cost derivation, both of which are orders of magnitude less than the time of making a what-if call—the $y$-axis of Figure 16 is in logarithmic scale. Second, the coverage-based refinement increases the computation time of the lower-bound, but it remains negligible compared to a what-if call.

Table 2 further presents the additional overhead of Wii w.r.t. the baseline configuration search algorithm without Wii, measured as a percentage of the baseline execution time. We observe that Wii’s additional overhead, with or without the coverage-based refinement, is around 3% at maximum, while the typical additional overhead is less than 0.5%.

### 6.5 Storage Constraints

As mentioned earlier, one may have other constraints in practical index tuning in addition to the cardinality constraint. One common constraint is the storage constraint (SC) that limits the maximum amount of storage taken by the recommended indexes [19]. To demonstrate the robustness of Wii w.r.t. other constraints, we evaluate its efficacy by varying the SC as well. In our evaluation, we fix $K = 20$, $\alpha = 0.9$, $B = 1,000$ for TPC-H and $B = 5,000$ for the other workloads, while varying the allowed storage size as $2\times$ and $3\times$ of the database ($3\times$ is the default setting of DTA [1]).

Figures 17 and 18 present the evaluation results for two-phase greedy and MCTS. Overall, we observe similar patterns in the presence of SC. That is, Wii, with or without the coverage-based refinement, often significantly outperforms the baseline approaches, especially for two-phase greedy.

### 6.6 Beyond Derived Cost

When Wii decides to skip a what-if call, it returns the derived cost (i.e., the upper bound) as an approximation of the what-if cost. This is not mandatory, and there are other options. For example,
one can instead return the average of the lower and upper bounds. We further evaluate this idea below. Figures 19 and 20 present the results. While both options perform similarly most of the time, we observe that they perform quite differently in a few cases; moreover, one may outperform the other in these cases. For example, with the coverage-based refinement enabled in Wii, when setting \( \alpha = 0.5 \), on TPC-H returning the average significantly outperforms returning the upper bound (74.7% vs. 59.7%); however, on Real-M returning the average loses 10.5% in percentage improvement compared to returning the upper bound (11.8% vs. 22.3%). As a result, the question of having a better cost approximation than the upper bound (i.e., the derived cost) remains open, and we leave it for future exploration.

6.7 Impact of Submodularity Assumption

Although our validation results show that submodularity holds with probability between 0.75 and 0.89 on the workloads tested [41], it remains an interesting question to understand the impact on Wii when submodularity does not hold. As we mentioned in Section 3.2.2, submodularity does not hold often due to index interaction [31]. For example, the query optimizer may choose an index-intersection plan with two indexes available at the same time but utilizing neither if only one of them is present. In this example, submodularity does not hold, because the MCI of either index will increase after the other index is selected. As a result, Equation 8 is no longer an MCI upper-bound—it will be smaller than the actual MCI upper-bound. Consequently, the \( L(q, C) \) computed
Table 3. Magnitude of violation (of submodularity).

<table>
<thead>
<tr>
<th>Workload</th>
<th>Average</th>
<th>Median</th>
<th>95th Percentile</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-H</td>
<td>0.209</td>
<td>0.001</td>
<td>1.498</td>
</tr>
<tr>
<td>TPC-DS</td>
<td>2.203</td>
<td>0.001</td>
<td>10.532</td>
</tr>
<tr>
<td>Real-D</td>
<td>7.658</td>
<td>0.010</td>
<td>38.197</td>
</tr>
<tr>
<td>Real-M</td>
<td>4.125</td>
<td>0.001</td>
<td>31.358</td>
</tr>
</tbody>
</table>

by Equation 4 will be larger than the actual lower-bound of the what-if cost, which implies an overconfident situation for Wii where the confidence is computed by Equation 10. The degree of overconfidence depends on the magnitude of violation of the submodularity assumption, which we further measured in our evaluation (see [41] for details).

Table 3 summarizes the key statistics of the magnitude of violation measured. Among the four workloads, we observe that Real-D and Real-M have relatively higher magnitude of violation, which implies that Wii tends to be more overconfident on these two workloads. As a result, Wii is more likely to skip what-if calls that should not have been skipped, especially when the confidence threshold $\alpha$ is relatively low. Correspondingly, we observe more sensitive behavior of Wii on Real-D and Real-M when increasing $\alpha$ from 0.5 to 0.9 (ref. Figures 9 and 12).

6.8 The Case of Unlimited Budget
As we noted in the introduction, Wii can also be used in a special situation where one does not enforce a budget on the index tuner, namely, the tuner can make unlimited number of what-if calls. This situation may make sense if one has a relatively small workload. Although Wii cannot improve the quality of the final configuration found, by skipping unnecessary what-if calls it can significantly reduce the overall index tuning time.

To demonstrate this, we tune the two relatively small workloads, namely TPC-H with 22 queries and Real-D with 32 queries, using two-phase greedy without enforcing a budget constraint on the number of what-if calls. We do not use MCTS as it explicitly leverages the budget constraint by design and cannot work without the budget information. We set $K = 20$ for TPC-H and $K = 5$ for Real-D in our experiments to put the total execution time under control. We also vary the confidence threshold $\alpha \in \{0.8, 0.9\}$ for Wii. Table 4 summarizes the evaluation results.

We observe significant reduction of index tuning time by using Wii. For instance, on TPC-H when setting the confidence threshold $\alpha = 0.9$, the final configurations returned by two-phase greedy, with or without Wii, achieve (the same) 85.2% improvement over the existing configuration. However, the tuning time is reduced from 8.2 minutes to 1.9 minutes (i.e., 4.3× speedup) when Wii is used. As another example, on Real-D when setting $\alpha = 0.9$, the final configurations returned, with or without Wii, achieve similar improvements over the existing configuration (64% vs. 62.3%). However, the tuning time is reduced from 380.6 minutes to 120 minutes (i.e., 3.2× speedup) by using Wii. The index tuning time on Real-D is considerably longer than that on TPC-H, since the Real-D queries are much more complex.

7 RELATED WORK

Index Tuning. Index tuning has been studied extensively by previous work (e.g., [4, 5, 7, 8, 12, 17, 20, 30, 35, 37, 40, 42, 46]). The recent work by Kossmann et al. [19] conducted a survey as well as a benchmark study of existing index tuning technologies. Their evaluation results show that DTA with the two-phase greedy search algorithm [7, 8] can yield the state-of-the-art performance, which has been the focus of our study in this paper as well.

Budget-aware Configuration Enumeration. Configuration enumeration is one core problem of index tuning. The problem is NP-hard and hard to approximate [6, 11]. Although two-phase greedy is

Table 4. Index tuning time with unlimited budget.

the current state-of-the-art [19], it remains inefficient on large and/or complex workloads, due to the large amount of what-if calls made to the query optimizer during configuration enumeration [19, 26, 33, 37]. Motivated by this, [46] studies a constrained configuration enumeration problem, called budget-aware configuration enumeration, that limits the number of what-if calls allowed in configuration enumeration. Budget-aware configuration enumeration introduces a new budget allocation problem, regarding which query-configuration pairs (QCP’s) deserve what-if calls.

**Application of Data-driven ML Technologies.** There has been a flurry of recent work on applying data-driven machine learning (ML) technologies to various aspects of index tuning [36], such as reducing the chance of performance regression on the recommended indexes [13, 48], configuration search algorithms based on deep learning and reinforcement learning [21, 28, 29, 32], using learned cost models to replace what-if calls [33, 37], and so on. While we do not use ML technologies in this work, it remains interesting future work to consider using ML-based technologies, for example, to improve the accuracy of the estimated coverage.

**Cost Approximation and Modeling.** From an API point of view, Wii returns an approximation (i.e., derived cost) of the what-if cost whenever a what-if call is saved. There have been various other technologies on cost approximation and modeling, focusing on replacing query optimizer’s cost estimate by actual prediction of query execution time (e.g., [2, 14, 16, 23–25, 27, 34, 38, 43–45, 47]). This line of effort is orthogonal to our work, which uses optimizer’s cost estimate as the gold standard of query execution cost, to be in line with previous work on evaluating index configuration enumeration algorithms [8, 19].

**8 CONCLUSION**

In this paper, we proposed Wii that can be seamlessly integrated into existing configuration enumeration algorithms to improve budget allocation and ultimately quality of the final index configuration found. Wii develops and leverages lower and upper bounds of the what-if cost to skip unnecessary what-if calls during configuration enumeration. Our evaluation results on both industrial benchmarks and real workloads demonstrate the effectiveness of Wii.

**Acknowledgments:** We thank the anonymous reviewers, Arnd Christian König, Anshuman Dutt, Bailu Ding, and Tarique Siddiqui for their valuable and constructive feedback. This work was done when Xiaoying Wang was at Microsoft Research.
REFERENCES


