ABSTRACT

Index tuning; i.e., selecting indexes that are appropriate for the workload to obtain good system performance, is a crucial task for database administrators. Administrators rely on automated index advisors for this task, but existing advisors work either offline, requiring a-priori knowledge of the workload, or online, taking the administrator out of the picture and assuming total control of the index tuning task. Semi-automatic index tuning is a new paradigm that achieves a middle ground: the advisor analyzes the workload online and provides recommendations tailored to the current workload, and the administrator is able to provide feedback to refine future recommendations. In this demonstration we present Kaizen, an index tuning tool that implements semi-automatic tuning.

Categories and Subject Descriptors
H.2.2 [Physical Design]: Access Methods

General Terms
Algorithms, Performance, Design

Keywords
semi-automatic, index advisor, feedback

1. INTRODUCTION

One of the crucial and challenging tasks that a database administrator (DBA) performs is index tuning, that is, selecting a set of indexes that are appropriate for the workload. This task is very hard for several reasons: there exists a large set of candidate indexes, some indexes might benefit some parts of the workload while at the same time incur very high maintenance costs (when the data is updated), and the benefit of an index may interact with the benefit of other materialized indexes.

In order to aid the DBA in this process, most commercial systems offer automatic index advisors (e.g., [1, 8, 7]). An index advisor takes as input a workload and some constraints (e.g., the total space of the materialized indexes is constrained by some budget), and generates a recommended set of indexes (also referred to as a recommended configuration) that results in the lowest execution cost for the given workload and satisfies the constraints.

Existing approaches to index tuning can be classified into two categories, namely offline and online techniques. Offline techniques (e.g., [2, 5]) generate a recommended configuration for a specific workload and let the DBA decide whether to create or drop indexes contained in it. In these approaches, the DBA is left with the non-trivial task of finding a good representative workload. Furthermore, this task becomes more challenging in dynamic environments when workload patterns evolve over time. At the other extreme, online techniques (e.g., [4, 9]) monitor the workload and automatically create or drop indexes autonomously. These approaches are essential to handle dynamic workloads, however, the “completely automatic” aspect of these tools does not leave any kind of control over the system’s performance, resulting in an adverse reaction by DBAs who are unlikely to favor this kind of tools.

In this paper, we introduce Kaizen\(^1\), a new index tuning tool that works in a semi-automatic fashion [11] to combine the best features of the offline and online paradigms. At a high level, Kaizen generates recommendations by analyzing the workload online, i.e., in parallel with query processing. This feature allows Kaizen to tailor its recommendations to the current traits of the workload, and also alleviates the DBA from the difficult task of providing a representative workload. Moreover, Kaizen has solely a consulting role, i.e., the DBA is ultimately responsible for scheduling the changes to the materialized indexes set. Finally, Kaizen provides a mechanism by which the DBA can provide feedback on the recommended indexes, which in turn affects the generation of subsequent recommendations. In this fashion, automatic workload analysis is coupled with human expertise in order to improve the effectiveness of index tuning. This is the feature that sets Kaizen apart from other advisors.

We note that most of existing index advisors are non-interactive: the normal process starts with a full input spec-
 Demonstration Structure. We present four index tuning scenarios that show how Kaizen implements the semi-automatic index paradigm. The first scenario familiarizes the audience with the tool’s interface and gives a high-level overview of the algorithm at the core of the tool. The second and third scenarios delve into the salient features of the algorithm: (1) a divide-and-conquer approach to online index tuning that leads to improved running time and better guarantees on recommendation quality; and (2) a principled feedback mechanism that is tightly integrated with the index-recommendation logic. The audience will witness how Kaizen “reacts” after receiving “good” or “bad” feedback. With good feedback, the recommendation quality increases immediately; with bad feedback, it decreases for a while but the algorithm is able to recover gracefully. The fourth and last scenario demonstrates how our tool can scale up and be used in the analysis of large datasets.

The remainder of the paper is organized as follows: Section 2 summarizes the salient features of the proposed algorithm for semi-automatic index tuning; Section 3 describes Kaizen’s architecture; and Section 4 presents the demonstration scenarios.

2. WFIT: A SEMI-AUTOMATIC INDEX TUNING ALGORITHM

At the core of Kaizen is WFIT [11], an online algorithm that provides an implementation of the semi-automatic paradigm. Formally, WFIT takes as input a workload stream \( Q \) and a feedback stream \( V \), and produces a stream of recommended index sets \( S \subseteq I \), generated after each element (query or feedback) in \( Q \cup V \). The computation of \( S \) can use information solely from past queries and votes, i.e. the algorithm has absolutely no information about the future.

A good semi-automatic tuning algorithm should recommend indexes that minimize the overall work done by the system, including the cost to process the workload as well as the cost to implement changes to the materialized indexes. The first component is typical for index tuning problems and reflects the quality of the recommendations. The second component stems from the online nature of the problem: the recommendations apply to the running state of the system, and it is clearly desirable to change the materialized set at a low cost.

Performance Metrics. WFIT bases its recommendations on a cost metric that captures the total work performed by the DBMS. Formally, we model the workload \( Q \) as a stream of queries and updates, with \( q_n \) denoting the \( n \)-th statement and \( Q_N \) denoting the prefix of length \( N \). Let \( S_n \) be the recommendation that an algorithm \( A \) generates after analyzing the statement \( q_n \) and all feedback up to \( q_{n+1} \). Let \( S_0 \) be the initial set of indexes. We define \( \text{totWork}(A, Q_N, V) \) as the total work performed by the DBMS under the recommendations generated by algorithm \( A \):

\[
\text{totWork}(A, Q_N, V) = \sum_{n=1}^{N} \text{cost}(q_n, S_n) + \delta(S_{n-1}, S_n).
\]

The value of \( \text{totWork}(A, Q_N, V) \) models the performance of a system where each recommendation \( S_n \) is adopted by the DBA for the processing of the statement \( q_n \). The first component corresponds to the cost of processing the workload. Here, \( \text{cost}(q_n, S_n) \) denotes the cost of executing the statement \( q_n \) using the index configuration \( S_n \). The second component is the transition cost and is associated to the cost of implementing the changes to the materialized set of indexes; i.e. how costly it is to go from configuration \( S_{n-1} \) to \( S_n \). We use this metric to evaluate WFIT’s performance in our demo.

Overview of WFIT. Figure 1(b) illustrates a high-level schematic of how WFIT works. WFIT first analyzes an incoming query to generate interesting indexes, which are added to the pool of candidate indexes. Subsequently, WFIT analyzes the interactions between candidate indexes. Two indexes \( a \) and \( b \) interact if the benefit of \( a \) depends on the presence of \( b \) and vice-versa. For example, \( a \) and \( b \) can interact if they are intersected in a physical plan, since the benefit of each index may be boosted by the other. WFIT uses index interactions to partition the candidates into subsets, such that indexes in different subsets do not interact. This is termed the stable partition and can be computed with the algorithm proposed in [12]. The bottom line is that indexes from different subsets can be selected independently.

In the next step, WFIT applies a divide-and-conquer strategy on the set of candidates. For each subset in the stable partition, WFIT uses a separate instance of WFA\(^+\) in order to track the benefit of recommending indexes from the specific partition. WFA\(^+\) is a variant of the Work Function Algorithm (WFA) [3] that is tailored to the index-tuning problem. By using this divide-and-conquer strategy, WFIT avoids the combinatorial explosion that would result if all indexes were put in one single partition.
3. ARCHITECTURE

The DBA may request the current recommendation at any point in time and provide feedback. Any feedback is incorporated back to each instance of WFA+ and considered for the next recommendation. The feedback is provided by casting positive votes for indexes in some set $F^+$ and negative votes for a disjoint set $F^-$. These sets ($F^+$, $F^-$) might involve any index, even indexes that are not part of the current candidate pool. WFIT provides a mechanism to consider the feedback ($F^+$, $F^-$) as if the workload (rather than the feedback) had led WFIT to recommend creating $F^+$ and dropping $F^-$. With this approach, WFIT can naturally recover from bad feedback if the future workload favors a different configuration.

Kaizen comprises three submodules:

DBTune Framework: The backend layer (Figure 1(a)) provides an abstraction to the functionality of the DBMS; it handles the communication with the database engine, gives access to object metadata and exposes a generic interface to execute what-if optimization calls. The DBTune framework allows Kaizen to operate on top of any DBMS, provided that two services are available: access to the what-if optimizer, and an implementation of candidate-index extraction for a single SQL statement. This design makes Kaizen easily portable; as these services are common primitives found in other systems (e.g., [1, 7]). For this demonstration we use IBM DB2 Express-C version 9.7 for 64-bit Linux.

Recommendation Engine: The recommendation layer (Figure 1(b)) is in charge of implementing the divide-and-conquer strategy behind WFIT: partitioning the candidate index set, assigning a WFA+ instance to each partition and generating index recommendations.

UI: The user interface (Figure 1(c)) is used by the DBA to interact with the tool. It allows the DBA to (1) inspect the system by providing access to its catalog; (2) provide feedback through the algorithm’s voting mechanism; and (3) quickly explore index recommendations. The UI is composed of a graphical and a command-line module. As noted in [6], DBAs favor text-based interfaces over graphical ones since they are easily customizable (through the usage of user scripts) and they allow access to more details about the tuning process.

The usage flow of Kaizen is as follows. The tool accepts as input the incoming workload, routing each statement to the DBMS and also to the recommendation engine. (We also provide a hook by which the user can request the analysis of a specific workload read from a file.) Through the GUI or CLI (Listing 1), the user can request for the status of the algorithm (lines 9-13) and drill-down in the generated recommendations (lines 15-19). Feedback can also be provided at any time (line 21).

4. DEMONSTRATION

We describe four scenarios that demonstrate the features of Kaizen. Scenarios 1-3 run over a small workload composed of three phases involving two queries and one update ($Q = \{q_1, u_2, q_3\}$). Each phase is composed of a few statements. The first phase consists only of repetitions of $q_1$; the second of repetitions of $u_2$; the third is a mix of $q_1$ and $q_3$. This “tiny suite” provides a context with uncomplicated content that the audience can follow throughout the description of Kaizen. Scenario 4 executes a large workload, demonstrating the ability of Kaizen to scale up and support the analysis of large datasets.

Scenario #1. Our first goal is to introduce the main ideas behind WFIT (the core of Kaizen) and to familiarize the audience with the user interface. We lay out a simplified scenario without user feedback where the set of candidate indexes is fixed (partitions don’t change as the algorithm runs). These constraints allow us to focus the description of the recommendation logic more easily; i.e. identify which indexes are created/dropped based on the statements that Kaizen observes. For the “tiny-suite” workload, we will find that the candidate set is small (only few indexes are in play for our small workload) and thus will be straightforward for the audience to understand what decisions the algorithm is making at each step.

As part of this high-level exposition of WFIT, we show a comparison of its performance against an optimal algorithm, referred to as OPT, that has full knowledge of the workload and generates the optimal recommendations that minimize the total work. In our case, OPT corresponds to the off-line algorithm implemented by the DB2 Advisor [7]. It can be replaced by any off-line recommender, such as [1] or [8]. Note that in practice, we cannot have an optimal baseline solution since the workload is typically not known in advance in the online context.

Listing 1: Sample CLI session invoking WFIT.

```
1: dbtune> var db = connect("postgresql://localhost")
2: dbtune> db.schemas
3: schemas = ["tpcc","tpcds","tpce","tpch","nref"]
4: dbtune> var wfit = WFIT("online-benchmark.sql")
5: Processing workload...
6: dbtune> wfit.status
7: WFITStatus =
8: partitions: 2
9: indexes: 3
10: statements_processed: 3
11: dbtune> wfit.recommendationAt(3)
12: Configuration =
13: 1: "INDEX sat_1 ON tpcc.orderline(cl_amount)"
14: 2: "INDEX sat_2 ON tpch.lineitem(l_commitdate)"
15: 3: "INDEX sat_3 ON tpce.daily_market(dm_close)"
16: dbtune> wfit.voteUp(2)
```

Figure 2: Comparing WFIT against OPT. Queries 1-5 correspond to phase one; 6-10 to phase two; phase three is not shown.
The UI is capable of graphing the ratio \( \frac{\text{totWork}_{\text{OPT}}}{\text{totWork}_{\text{WFIT}}} \) (Figure 2) in parallel with the analysis of the query stream. A high ratio indicates that WFIT generates good recommendations. It also makes available the recommendations that are generated at each step, as well as the internal bookkeeping that the algorithm maintains. We will show some of this information as part of this scenario.

**Scenario #2**. We delve a little bit more into the details of our tool by allowing the candidate-index set to be automatically maintained but again keeping the feedback feature “off”. At this point, the candidate-index set can dynamically grow/shrink and be repartitioned over time based on the calculations of index interactions associated with each statement. This brings the tool into a completely online mode where it can operate autonomously without any user intervention.

![Candidate Set Evolution](image)

**Figure 3**: Evolution of the candidate set with respect to partitioning (by calculating index interactions at each step). Each set corresponds to phases 1, 2 and 3 respectively.

We will see again how the algorithm generates a configuration at each step, however, in this scenario the partitioning of the candidate set will evolve for each of the three phases of the workload (Figure 3). We will show that this feature actually improves the quality of the recommendations.

**Scenario #3**. We complete the picture and show the effect that feedback has on the performance of WFIT by demonstrating one of the key contributions of our work: a principled feedback mechanism that is tightly integrated with the logic of the on-line algorithm (WFA*).

By inspecting the recommended set of indexes at any point in time, the DBA can decide whether to up- or downvote any candidate index according to her criteria (or not vote at all). For the small test workload, it is easy to come up with reasonable “good” and “bad” votes that the audience can interactively send as feedback and show the difference in performance for each (Figure 4).

The audience will see how, in the case of “good” feedback, the performance of WFIT increases in relation to the performance of the “no-feedback” instance (using the performance of OPT as baseline). In contrast, with “bad” feedback, the performance of WFIT will decrease; however, and more importantly, we will witness how WFIT is able to recover from poor feedback. This recovery mechanism is another important feature of the WFIT algorithm.

**Scenario #4**. The last scenario executes the Reflex workload suite of the Online Index Selection Benchmark [10] on Kaizen. This is a complex workload consisting of approximately 1600 statements (queries and updates) that reference several datasets (TPC-C, TPC-DS, TPC-E, TPC-H and NREF).

We will show two WFIT variants: one with a stable and fixed candidate set partitioning; another whose candidate set is allowed to be automatically maintained. Similarly to scenario #1, we will graph the OPT vs. WFIT ratio in real-time as the workload is processed (Figure 5).

![Algorithm Comparison](image)

**Figure 4**: Multiple instances of WFIT running in parallel. The vote for the “good” and “bad” instances is done at step 1, causing the divergence in their behavior with respect to the “no-feedback” instance.

![Algorithm Comparison](image)

**Figure 5**: Two instances of WFIT running the Online Index Selection Benchmark. One with a fixed and stable candidate set (FIXED); another one with an automatically maintained candidate set (AUTO).

5. REFERENCES


