On-Line Index Selection for Shifting Workloads

Karl Schnaitter[†] Serge Abiteboul[‡] Tova Milo[±] Neoklis Polyzotis[†]

[†]Univ. of California Santa Cruz
{karlsch,alkis}@cs.ucsc.edu

[‡]INRIA and Univ. Paris 11
fname.lname@inria.fr

[±]University of Tel Aviv
milo@cs.tau.ac.il

Abstract

This paper introduces COLT *(Continuous On-Line Tuning), a novel framework that continuously monitors the workload of a database system and enriches the existing physical design with a set of effective indices. The key idea behind* COLT *is to gather performance statistics at different levels of detail and to carefully allocate profiling resources to the most promising candidate configurations. Moreover,* COLT *uses effective heuristics to self-regulate its own performance, lowering its overhead when the system is well tuned and being more aggressive when the workload shifts and it becomes necessary to re-tune the system. We describe an implementation of the proposed framework in the PostgreSQL database system and evaluate its performance experimentally. Our results validate the effectiveness of* COLT *and demonstrate its ability to modify the system configuration in response to changes in the query load.*

1. Introduction

One of the main tasks of a database administrator involves tuning the physical schema of the system, that is, installing physical access structures, such as indices or materialized views, that assist the data server in optimizing the query load more efficiently. The selection of these access structures is itself an optimization problem: the administrator must optimize the system throughput, assuming some limited storage resources for the materialized structures.

To assist the administrator in this challenging task, earlier studies [4, 8, 23] have introduced techniques that analyze a representative workload and automatically generate a recommended physical configuration. This paradigm is typically referred to as *off-line tuning*, since the workload is gathered and analyzed before the database system goes live. The use of a representative workload implies that off-line tuning is suitable for the stable component of a query load, that is, the subset of query characteristics that are predominant. On the other hand, one can identify an unstable component that, while not being present in most of the queries in the whole workload, it is present in most of the queries in some contiguous subset of the workload. One such example are workloads that result from interactive data analysis. In this context, a user issues exploratory queries to validate various hypotheses against the data, and the consecutive queries related to a single hypothesis may have similar characteristics.

The previous discussion hints at the desirability of an *on-line* tuning mechanism that complements the off-line tuning of a database system. The main idea is to monitor the query load to identify locally dominant patterns (vs. the globally dominant patterns identified by off-line tuning), and automatically adjust the physical configuration to maximize query performance.

Prior Work. A large subset of previously proposed on-line tuning techniques fall in the general area of semantic caching [10], and its variants [14, 15] that are specialized to specific domains. We can identify Cache Investment [13], QUIET [17], and the work of Hammer and Chan [12] as the techniques most relevant to the on-line tuning of the physical configuration of a DBMS. These techniques focus on the problem of automatic index selection and adopt the same working model: the system continuously monitors the workload to identify candidate indices, it profiles their benefit, and materializes the most promising ones. The profiling is typically performed through a *what-if* optimizer, which essentially optimizes queries (and thus computes their cost) assuming that candidate indices are materialized. A key issue, however, is that the aforementioned works do not provide an explicit mechanism to regulate the issuance of what-if calls. Thus, the on-line process operates with the same intensity even if the system cannot be tuned to work better for the current workload, and it is not straightforward to control the number of what-if calls used by on-line tuning. These two points are critical for controlling the overhead of on-line tuning, and are thus key for its incorporation in real-world systems.

There has also been a large body of research in off-line methods for database tuning. As the name suggests, off-line techniques work outside of the continuing operation of the database system. Typically, such tuning methods employ a representative query load in order to both generate candidate physical configurations and to evaluate their effectiveness. Earlier studies on this topic include techniques for the automatic selection of indices [8, 11] and/or materialized views [1, 2, 3, 4], while several commercial systems include auto-tuning tools [7, 23] that are based on the off-line approach. Off-line methods, however, are not well suited for on-line tuning since their cost is prohibitive for the continuous monitoring of the query load. Moreover, they do not incorporate any mechanisms for allocating the use of profiling resources or self-regulating the overhead of tuning – these are clearly non-issues as all computation is performed off-line. A recent work has developed a physical design alerter [5], which essentially approximates the result of an off-line tuning tool in a fraction of the time. Given a set of queries as input, the alerter provides upper and lower bounds on the performance improvement that would be possible with a comprehensive tuning tool.

Our Contributions. In this paper, we introduce the COLT framework (short for *Continuous On-Line Tuning*) that supports the automatic on-line materialization of index structures on top of an existing pre-tuned configuration. COLT builds a model of the current workload based on the incoming flow of queries, estimates the respective gains of different indices, and selects those that would provide the best performance for the observed workload within the available space constraint. The novel feature of COLT compared to previous works is that it incorporates an explicit mechanism for controlling its overhead. Thus, the administrator can specify the maximum rate of what-if calls issued by COLT, but most importantly, COLT is able to self-regulate its performance. Intuitively speaking, the latter amounts to decreasing the intensity of on-line tuning when the system is well tuned, and increasing it when a phase shift occurs in the workload. To the best of our knowledge, ours is the first work on on-line tuning to address this important issue of controllable overhead.

We also present the results of a preliminary experimental study based on a prototype implementation inside the PostgreSQL database server. Our results show that COLT discovers an effective set of materialized indices and adapts rapidly to shifts in the query distribution. Moreover, it is able to control the overhead of on-line tuning and regulate it according to the changes in the query load.

2. Problem Definition

At an abstract level, the tuning (optimization) problem that we consider in this paper may be described as follows: given a pre-tuned physical database design, the current query distribution $\mathcal{Q}$, and an on-line storage budget of B units, select the set $\mathcal{I}$ of additional indices that minimize the expected query evaluation cost and fit in the allotted storage budget. Thus, $\mathcal{I}$ is expected to vary over time in order to adapt to changes in $\mathcal{Q}$. We also note that the query distribution is not observable, so it must be guessed from past queries.

Similar to off-line tuning algorithms, a solution to this on-line tuning problem must rely on what-if calls to the query optimizer to select access structures that match the underlying query execution cost model. Moreover, the on-line algorithm must address the following important issues that are unique to continuous tuning.

- *Adaptivity and Resilience to Noise.* In order to be effective, the tuning process must adapt relatively fast to changes in the query load. At the same time, it should not overreact to temporary variations of the query distribution but focus instead on real changes of the workload.

- *Adaptive Overhead.* Since the tuning process runs concurrently with normal query processing, it is important to maintain a controlled overhead in order to avoid penalizing normal query execution. Furthermore, the overhead the system is willing to spend on tuning should depend on the gain that is expected from a modification of the physical schema.

These issues introduce interesting trade-offs that are not trivial to balance. For instance, the tight coupling with the query optimizer hints at additional optimizer invocations, which in turn increase the cost of self-tuning. Our observations also indicate that adopting existing off-line methods is not a feasible solution to the problem. In particular, off-line methods use a specific instance of the workload and thus behave well only if the actual load is rather stable in time. Moreover, they do not address adequately the issue of low overhead as they are executed separately from the running system. Hence, they typically require plenty of CPU resources and they can cause a serious slowdown if deployed on-line.

In our work, we tackle the variant of the problem where $\mathcal{I}$ consists of single-column indices. This may seem limiting at first, especially when compared to existing off-line tuning techniques that consider multi-column indices (clustered or unclustered) and materialized views. As we show in this paper, however, the transition to on-line tuning poses significant challenges that are not trivial to address even in this seemingly simplified setting. Moreover, a recent study [9] has shown that a set of carefully chosen single-column indices can offer significant improvements in query performance. The extension of our techniques to more general access structures, e.g., multi-column indices and materialized views, is an interesting direction for future work.

3. Overview of COLT

In this section, we present an overview of the proposed COLT framework for *continuous on-line tuning*. COLT di-

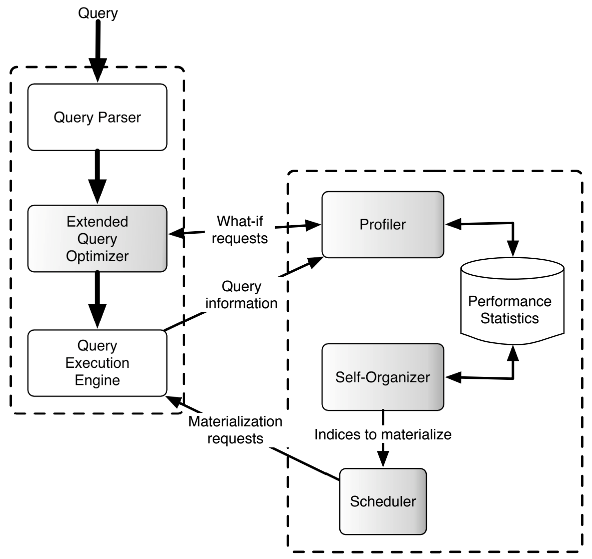

Figure 1. Architecture of COLT.

vides the incoming workload in non-overlapping windows of w queries, called *epochs*, where w is a system parameter. We use $\mathcal{S}_i$ to denote the sequence of queries in the most recent i epochs. During an epoch, COLT *profiles* each candidate index I on the corresponding queries in order to evaluate its potential benefit on the current query load. Thus, the measurements for I in recent epochs provide a picture of its potential performance as time progresses. At the end of an epoch, COLT initiates a *reorganization* phase that determines which indices should be materialized based on the performance statistics gathered while profiling. This continuous alternation between profiling and reorganization enables COLT to track the current workload and adapt the physical configuration accordingly.

COLT maintains a set $\mathcal{C}$ of *candidate indices* by mining the selection predicates of queries in the sequence $\mathcal{S}_h$. Here, h is a global parameter that regulates the extent of the system's memory and should be large enough in order to capture the dominant traits of the query workload. COLT continuously profiles the indices in $\mathcal{C}$ and carefully selects a subset $\mathcal{M} \subseteq \mathcal{C}$, termed the *materialized set*, that is materialized and used for query evaluation. To avoid the high cost of extensive profiling for every candidate index, COLT employs a two level strategy. More precisely, each index in $\mathcal{C}$ is profiled with very easy to compute, yet crude performance statistics. These crude statistics are used to rank candidates and identify a small set $\mathcal{H} \subseteq \mathcal{C}$ of *hot* indices, that is, indices that have not been materialized but look promising for the current workload. COLT subsequently profiles hot and materialized indices with accurate and therefore more expensive methods, and from that derives the new materialized set at the end of the epoch.

Figure 1 presents an architectural diagram of COLT. As shown, COLT works in parallel to the main processing pipeline. The functionalities of its three main components can be summarized as follows:

• *Extended Query Optimizer (EQO).* The EQO extends a standard query optimizer by providing a *what-if* optimization interface. More precisely, we assume that the optimizer exports a function call WHATIFOPTIMIZE$(q, \mathcal{P})$, where q is a query and $\mathcal{P}$ is a set of indices[1]. For each index $I \in \mathcal{P}$, the optimizer computes and returns the reduction in the execution cost of q assuming that I is materialized. Overall, the what-if optimizer constitutes the primary mechanism by which COLT measures accurately the effect of different indices on query evaluation. This tight coupling with the query optimizer is an essential element of our approach.

• *Profiler.* This component is responsible for gathering performance statistics for candidate indices. These statistics are updated incrementally after the evaluation of each query. As explained previously, the level of detail for the collected statistics changes depending on the set ($\mathcal{C}$, $\mathcal{H}$, or $\mathcal{M}$) of the candidate index. For $\mathcal{C}$, the Profiler maintains very crude performance statistics; for $\mathcal{H}$ and $\mathcal{M}$, the indices are profiled through the what-if interface of the EQO.

• *Self Organizer (SO).* This component implements the reorganization phase of COLT and is thus activated only at the end of each epoch. SO mines the performance statistics gathered by the Profiler and forecasts the expected benefit of each index in $\mathcal{H} \cup \mathcal{M}$ on the query workload. The indices with the highest expected benefits are then materialized, forming the new $\mathcal{M}$. The SO is also responsible for selecting the hot indices from $\mathcal{C}$ to have them profiled accurately in the coming epochs.

• *Scheduler.* All modifications to $\mathcal{M}$ are handled by the Scheduler. This gives COLT the power to choose the best time to materialize an index. Several scheduling strategies are possible, including (1) carrying out materialization requests immediately, (2) building indices during system idle time, and (3) using intermediate results of future queries to build indices more efficiently. In our implementation of the COLT framework, we take the first approach and issue immediate asynchronous requests (in parallel) for the indices that the Self Organizer selects for materialization. This is the most straightforward design, and it has the advantage that new indices are available as soon as possible.

We discuss the details of the Profiler and the Self Organizer in the following two sections.

[1] It is interesting to note that this what-if interface is already implemented in most commercial systems and is thus readily available.

4. Profiler

The Profiler is closely coupled with the EQO and its main function is to measure the performance of candidate indices. In our work, we measure the performance of an index in a specific epoch as the average reduction in execution time for the corresponding queries. More formally, consider a particular query q and some particular time t, and let $QueryCost(q, \mathcal{I})$ denote the optimal cost of evaluating q using the physical design that consists of the pre-tuned configuration and the single-column indices in $\mathcal{I}$. Let $QueryGain(q, I) = QueryCost(q, \mathcal{M} \cup \{I\}) - QueryCost(q, \mathcal{M} - \{I\})$ denote the savings in execution time of q when I is part of the materialized set $\mathcal{M}$ (with t understood). The benefit for an index I in the current epoch $\mathcal{S}_1$ is defined as the average $Benefit(I) = (\sum_{q \in \mathcal{S}_1} QueryGain(q, I))/w$.

Clearly, the exact computation of this metric would require a prohibitive cost in terms of *what-if* calls to the query optimizer. To obtain reasonable estimates of $Benefit(I)$ at moderate cost, COLT employs a two-level strategy. At the first level, the Profiler computes a crude approximation $Benefit_{\mathcal{C}}(I)$ of $Benefit(I)$ that is used to select the most promising candidate indices and place them in the hot set $\mathcal{H}$. At the second level, the Profiler uses what-if optimization calls to compute much better approximations $Benefit_{\mathcal{H}}(I)$ and $Benefit_{\mathcal{M}}(I)$ of $Benefit(I)$ for hot and materialized indices respectively. The key idea, therefore, is that $Benefit_{\mathcal{C}}$ is used for all candidates in $\mathcal{C}$, while $Benefit_{\mathcal{H}}$ and $Benefit_{\mathcal{M}}$ are directly comparable and used to compare indices between $\mathcal{H}$ and $\mathcal{M}$. The aforementioned approximations of benefits are computed with appropriate approximations of $QueryGain(q, I)$ for each query q, namely $QueryGain_{\mathcal{C}}(q, I)$, $QueryGain_{\mathcal{H}}(q, I)$, and $QueryGain_{\mathcal{M}}(q, I)$, respectively. We describe these approximations in the following section, and then discuss the profiling algorithm in more detail.

4.1. Gain estimation

***QueryGain$_{\mathcal{C}}$*.** The model for $QueryGain_{\mathcal{C}}$ relies on standard cost formulas to obtain an optimistic approximation of the true query gain. More formally, let q be a query in the current workload and $I \in \mathcal{C}$ be a relevant candidate index. Let R denote the table on which I is defined and σ be the selection predicate in q that I may help evaluate. We define a binary indicator variable $u_{q,I}$ that takes the value 1 if the optimizer uses I in the evaluation plan of q. This information can be derived from the normal optimization of q and any additional what-if calls if $I \in \mathcal{M} \cup \mathcal{H}$; for any other case, our system makes an optimistic prediction and sets $u_{q,I} = 1$. The Profiler approximates the gain of q as $QueryGain_{\mathcal{C}}(q, I) = u_{q,I} \cdot \Delta cost(R, \sigma, I)$, where $\Delta cost(R, \sigma, I)$ is a crude estimate of the gain in evaluating σ using I vs. using a sequential scan of R. (We use standard cost formulas [20] for this computation.)

Clearly, the resulting approximation $Benefit_{\mathcal{C}}$ is a crude estimate of the true performance of an index. The goal, however, is only to identify the most promising candidate indices in order to profile them more accurately at the next level. In this respect, we have found the previous metric to work adequately well.

***QueryGain$_{\mathcal{H}}$*.** As mentioned earlier, the Profiler relies on what-if optimization calls to obtain detailed performance metrics for indices in $\mathcal{H}$. To control the overhead of what-if calls, the Profiler relies on an intuitive assumption: for queries that are "similar", one is likely to observe similar benefits for a given index I. Hence, profiling I against a sample of queries only may provide enough information for the complete query set. More precisely, the Profiler maintains a clustering $Q_1, \ldots, Q_K$ of query occurrences in $\mathcal{S}_h$, where each cluster corresponds to a subset of queries that access the same tables, have the same join predicates, and have selection predicates on the same attributes with selectivity factors in specific ranges. We use two possible ranges of selectivity factors, namely, 0-2%, and 2-100%, yielding an approximate separation between selective and non-selective predicates. Each cluster Q_i records a count $Count(Q_i)$ of the queries that it represents, and a set of statistics (explained further) on the gains of indices for queries in Q_i. It is important to note that each query belongs to a unique cluster and that this assignment is performed efficiently on-line when the query arrives. Moreover, the system does not need to maintain a large number of clusters. In the worst case, the number of clusters can be equal to $w \cdot h$, the total number of queries in the system's memory.

The key idea behind our clustering model is to capture query similarity in the current workload and to maintain aggregate index statistics per cluster. Given an index $I \in \mathcal{H}$ and a related cluster Q_i, the Profiler measures $QueryGain$ accurately (through what-if calls) on a sample of Q_i, and uses the collected measurements to build a *confidence interval* $[LowGain(I, Q_i), HighGain(I, Q_i)]$ for the average gain of a query in Q_i with respect to I. In our work, we use CLT-style bounds [21] with a fixed level of confidence (e.g., 90%) for all pairs (I, Q_i). Typically, we would expect the interval to get reduced when more queries in Q_i are selected to be profiled. Given the similarity of queries in Q_i, however, it is highly likely to obtain a tight approximation with only a few what-if calls.

Based on these statistics, the Profiler computes $QueryGain_{\mathcal{H}}(q, I)$ (and hence $Benefit_{\mathcal{H}}(I)$) as follows. Let q be some query in cluster Q_i for some i. If q has been profiled against I then the Profiler already has knowledge of its gain and sets $QueryGain_{\mathcal{H}}(q, I) = QueryGain(q, I)$. Otherwise, the Profiler makes a conservative estimate and uses the lower bound $LowGain(I, Q_i)$ of the confidence interval as the approximated gain. Clearly, this method yields a conservative estimate as it underestimates the true query gain. This property is nonetheless desirable in our setting, as an index will be selected for materialization only if there is strong evidence of its good performance, i.e., when the *conservative* estimate indicates it is worth materializing.

As a final remark, we note that our approximation models for $QueryGain_{\mathcal{H}}$ rely on selective $QueryGain$ measurements that are time-sensitive by definition, i.e., they capture the difference in execution cost if an index is inserted (or dropped) from the contents of $\mathcal{M}$ at the time of measurement. Statistics may therefore become invalid as the set of materialized indices evolves over time. To address this issue, $QueryGain_{\mathcal{H}}$ only uses statistics that are consistent with the current materialized indices. In our context, a past measurement for a hot index is consistent if the relevant indices on the same table have not changed in $\mathcal{M}$.

***QueryGain*$_{\mathcal{M}}$.** As mentioned earlier, $QueryGain_{\mathcal{M}}$ is computed in a similar fashion as $QueryGain_{\mathcal{H}}$, i.e., by maintaining benefit statistics on a per *(cluster,index)* basis. The key difference is that the statistics capture the average *positive* benefit per query in the cluster, instead of the average overall benefit. This distinction is necessary since a materialized index I is always considered for the optimization of a query q, and hence COLT needs to approximate the true $QueryGain(q, I)$ only when I is used in the plan.

As with $QueryGain_{\mathcal{H}}$, the value of $QueryGain_{\mathcal{M}}$ is based on what-if calls, but the EQO handles materialized indices differently. This is because the normal optimization of a query will yield the execution cost with all materialized indices available. The what-if optimizer must pretend that a materialized index is *unavailable*, and the $QueryGain$ is given by the resulting *increase* in execution cost. This process is the reverse of traditional what-if optimization for indices that are not materialized.

4.2. Profiling Algorithm

Having introduced our estimation models for $QueryGain$ we now describe the algorithm for gathering the corresponding statistics in the course of an epoch. Figure 2 shows the pseudo-code for one invocation of the Profiler on the current query. After receiving the current query q and its initial optimized plan, the Profiler assigns q to a unique cluster Q_i and updates the statistics of the materialized and hot indices that are related to Q_i. For each such index I, it computes a sampling probability (function GETSAMPLERATE(I, Q_i)) for measuring $QueryGain(q, I)$ through a what-if call, and samples the indices with this probability. The sampled indices are included in the probation set $\mathcal{P}$ that is sent to the Extended Query Optimizer, and the returned query gains are used to update the corresponding interval statistics. We detail the computation of sampling probabilities later; the key intuition is to sample more from the (I, Q_i) combinations for which the gain estimate is believed to be very imprecise. Finally, the Profiler completes the processing of q by updating the statistics of $Benefit_{\mathcal{C}}(I)$ for each relevant candidate $I \in \mathcal{C}$ (line 14).

```
Procedure PROFILEQUERY($q$,$plan$)
Input: Current query $q$; its optimal physical plan $plan$.
begin
1.  Let $Q_i$ be the cluster of $q$
    /** Form probation set $\mathcal{P}$ **/
2.  Let $\#WI_{cur}$ be the number of performed what-if calls
3.  Let $\mathcal{I}_{\mathcal{M}} \subset \mathcal{M}$ be the materialized indices used in $plan$
4.  Let $\mathcal{I}_{\mathcal{H}} \subset \mathcal{H}$ be the hot indices relevant to $Q_i$
5.  $\mathcal{P} \leftarrow \emptyset$
6.  for each $I$ in a permutation of $\mathcal{I}_{\mathcal{M}}$ then $\mathcal{I}_{\mathcal{H}}$ do
7.      if $\#WI_{cur} + |\mathcal{P}| < \#WI_{lim}$ then
8.      with probability GETSAMPLERATE$(I, Q_i)$ add $I$ to $\mathcal{P}$
    /** Call what-if optimizer **/
9.  $QG \leftarrow$ WHATIFOPTIMIZE$(q, \mathcal{P})$
    /** Update statistics for $Benefit_{\mathcal{H}}$ and $Benefit_{\mathcal{M}}$ **/
10. for each $\langle I, QueryGain(q, I)\rangle$ in $QG$ do
11.     Update statistics for $(I, Q_i)$
12. $\#WI_{cur} += |\mathcal{P}|$
    /** Update statistics for $Benefit_{\mathcal{C}}$ **/
13. for each $I \in \mathcal{C}$ relevant to $q$ do
14.     Update $Benefit_{\mathcal{C}}(I)$ estimate with $QueryGain_{\mathcal{C}}(q, I)$
end
```

Figure 2. Profiling Algorithm.

Controlling the profiling overhead. At the beginning of each epoch, the Profiler is allocated a current profiling "budget" $\#WI_{lim}$ that bounds the number of what-if calls performed during the epoch. As will be explained in Section 5, this $\#WI_{lim}$ budget is set adaptively by the Self-Organizer based on the potential of the current hot indices. In a nutshell, the Self-Organizer increases $\#WI_{lim}$ if there is evidence that the current hot indices can lead to a much better materialized set, and conversely limits the budget (and even possibly sets it to 0) in the opposite case. The provided budget never exceeds a system parameter $\#WI_{max}$ that controls the maximum profiling overhead of COLT. A subtle point of the profiling algorithm is that materialized indices are given precedence in the spending of the what-if budget (line 6), as it is important to maintain accurate statistics for the materialized set.

Computing the sample rate. An important element of our approach is the use of adaptive sampling. We now describe in more detail the computation of the sampling probabilities through function GETSAMPLERATE(I, Q_i). For simplicity, we focus our discussion on the case of $I \in \mathcal{H}$. We use a similar method for $\mathcal{M}$.

Intuitively, a pair (I, Q_i) should be allocated more what-if calls if its profiling may have an important impact on the accuracy of the estimate $\mathit{Benefit}_{\mathcal{H}}(I)$. More precisely, this heuristic is captured as follows. As we show in the full version of the paper, each pair (I, Q_i) contributes independently to the error (in a statistical sense) of the estimate $\mathit{Benefit}_{\mathcal{H}}(I)$. Moreover, the error contribution grows with the popularity of Q_i and with the variance of profiled gains of I within Q_i, and shrinks as more of Q_i is profiled against I. A natural heuristic, therefore, is to allocate what-if calls in proportion to this error contribution, in order to profile more aggressively the inaccurate (I, Q_i) pairs and thus try to improve the accuracy of the estimated index benefit. Thus, we put the focus on indices that have the most unstable (or unpredictable) performance on the current workload. It is important to stress that the allocation is performed on-line: the Profiler reevaluates the error contributions after the what-if calls for (I, Q_i), which leads essentially to a modified sampling probability for (I, Q_i).

5. Self Organizer

The Self Organizer component is invoked at the end of each profiling epoch and essentially performs two operations: (a) it determines the new composition of the hot and materialized sets based on the performance statistics of the existing indices in $\mathcal{C}$, and (b) it sets the profiling budget $\# WI_{lim}$ based on the potential benefit of the currently hot indices. We refer to these operations as *reorganization* and *re-budgeting* resp., and discuss them in more details further.

Reorganization. The SO first chooses the new composition of the materialized set $\mathcal{M}$ by selecting the most promising indices from $\mathcal{H} \cup \mathcal{M}$. It then designates a subset of the remaining indices as the new hot set $\mathcal{H}$. We first describe the selection of $\mathcal{M}$.

Indices are chosen for materialization based on a metric that predicts the net benefit of indices in the near future. The system uses the statistics from the past h epochs to predict the benefit for the next h epochs (recall that h is the number of epochs in the system's memory). The net benefit takes into account improvements in query execution time and the cost of index materialization. The metric is defined as $\mathit{NetBenefit}(I) = \sum_j \mathit{PredBenefit}_j(I) - \mathit{MatCost}(I)$. $\mathit{MatCost}(I)$ is an estimate of the cost of materializing I (if I is already materialized, $\mathit{MatCost}(I) = \mathit{0}$). $\mathit{PredBenefit}_j(I)$ is the forecasted benefit that I will have for query execution in future epoch j. The value is computed taking all of the past j epochs into account. The complete details of the forecasting function can be found in the full version of this paper [19]. The new materialized set is formed by solving an instance of the KNAPSACK problem: the set of objects is $\mathcal{H} \cup \mathcal{M}$; the size of the knapsack is the storage budget B; each object I occupies $\mathit{IndexSize}(I)$ units of storage, and provides $\mathit{NetBenefit}(I)$ units of value. The KNAPSACK model is not completely accurate because the benefits of different indices are not always independent. However, our preliminary experiments indicate that this model can work well in this context.

We also note that previously proposed methods for offline tuning have used the KNAPSACK model for the selection of physical structures [6, 22]. Our use of the model differs in two ways. First, the value of an index is given by the $\mathit{NetBenefit}$ metric that predicts the benefit of an index, taking the materialization cost into account. Second, the SO determines a new KNAPSACK solution at the end of each epoch. This allows the system to correct some mistakes that may result from the inaccuracy of the model. For example, suppose a materialized index I becomes useless due to some change in the materialized set. The SO should remove I from $\mathcal{M}$, but the KNAPSACK model does not make this apparent, since the benefit of indices are assumed to be independent. However in future epochs, I will be unused and its predicted benefit will converge to zero. This means that I will not be included in the optimal KNAPSACK solution, and it will be dropped from the materialized set.

The second stage of the reorganization task selects the hot set for the next epoch. The SO computes a smoothed average of the crude $\mathit{Benefit}_{\mathcal{C}}(I)$ estimates for each of the remaining candidate indices, and groups the benefit estimates into two clusters with minimum variance. The indices in the top cluster are considered the most promising according to $\mathit{Benefit}_{\mathcal{C}}$, and are selected as the new hot set.

Re-budgeting. The goal of re-budgeting is to compute an appropriate value for the what-if budget $\# WI_{lim}$ of the upcoming epoch. Our strategy is to intensify profiling if the hot indices are likely to be more beneficial than the already materialized indices, and to decrease profiling, or even suspend it, in the opposite case. This mechanism enables self-tuning to "hibernate" when the workload is stable and $\mathcal{M}$ performs well, and to wake up when a shift occurs and new indices need to be materialized.

To asses the potential of the currently hot indices, the Self Organizer considers a best-case scenario for their performance. More concretely, it adjusts $\mathit{Benefit}_{\mathcal{H}}(I, Q_i)$ to utilize the upper bound of the confidence interval and thus computes an overestimate of $\mathit{Benefit}(I)$; this is in turn used to compute an overestimate of $\mathit{NetBenefit}(I)$ for each hot index. (Note that the metrics for materialized indices are left untouched.) Based on these optimistic predictions for hot indices, the Self Organizer solves the KNAPSACK problem again and computes another composition $\mathcal{M}'$ for the materialized set. The idea is to compare $\mathcal{M}$ and $\mathcal{M}'$ in terms of their projected performance and to determine accordingly how to set $\# WI_{lim}$. More precisely, we use the aggregate $\mathit{NetBenefit}$ metric of each set and compute the ratio $r = \mathit{NetBenefit}(\mathcal{M}')/\mathit{NetBenefit}(\mathcal{M})$. (Note that $r \geq 1$.) We adopt a scheme that suspends profiling if $r = 1$ and maximizes it to $\#WI_{max}$ if $r \geq 1.3$.

6. Experimental Study

We next present the results of an empirical study that we have conducted to evaluate the performance of COLT. Our study focuses on the following aspects of COLT: its ability to choose an effective set of indices and to adapt to the current workload; the resilience to noise in the workload; and the overhead of on-line tuning. We detail our experimental methodology and the main results of our study.

6.1. Methodology

In this section, we describe the self-tuning techniques that we evaluated in our study, the data sets and corresponding workloads, and the evaluation metrics.

Self-Tuning Techniques. We base our experimental study on two self-tuning techniques, namely, COLT and an optimal off-line technique.

- COLT. We implemented a prototype[2] of COLT inside the PostgreSQL database system [16]. The prototype follows the architecture presented in Section 3. It extends the Postgres optimizer with a what-if interface and features COLT as a separate sub-process of the Postgres server. In order to minimize the overhead of what-if calls, our implementation of the what-if optimizer reuses intermediate solutions from the initial query optimization. Of course, not all intermediate solutions can be reused, so the EQO must be careful only to reuse the solutions to subproblems that do not depend on the index being evaluated.

We use the following values for the system parameters: epoch length $w = 10$; history depth $h = 12$; maximum number of what-if calls $\#WI_{max} = 20$; confidence level of intervals 90%. We note that the results we present were not sensitive to the exact values of these parameters.

- OFFLINE. We have implemented an off-line tuning technique, henceforth referred to as OFFLINE. OFFLINE has knowledge of the exact workload and examines *exhaustively* the space of all possible single-column index sets, evaluating the effectiveness of each candidate with the same what-if optimizer that is used by COLT. The returned configuration is thus optimal with respect to the specific workload and the allotted space budget. Clearly, OFFLINE represents an idealized off-line technique as it has complete knowledge of the workload and infinite processing time. Thus, in the context of single-column index selection, it strictly dominates existing off-line techniques [4, 8] that rely on a heuristic search of the same configuration space.

Size (binary data)	1.4 GB
# Tables	32
# Tuples in all tables	6,928,120
# Tuples in largest table	1,200,000
# Tuples in smallest table	5
# Indexable attributes	244

Table 1. Data Set Characteristics

Data Set. We use a synthetic data set based on the well-known TPC-H schema. Table 1 summarizes its characteristics. The data set consists of 4 different data instances of the TPC-H schema, thus containing a large number of indexable attributes. This allows us to generate workloads that shift their focus between multiple tables and attributes.

We use synthetic query workloads that are randomly generated based on query distributions. We consider a workload with a fixed distribution, as well as workloads with a query distribution that changes over time. Details on the characteristics of individual workloads are given in the description of each experiment.

Evaluation Metric. We use the *total query execution time* as the basic metric for measuring the performance of a tuning technique. For OFFLINE this does not include the time spent to select and materialize an index set, since both tasks are assumed to take place off-line. For COLT, on the other hand, the measured query execution time is affected by the initially empty index set, the overhead of on-line tuning, i.e., the what-if calls and the index materialization. We measure query response time at the server, using a cold cache and a single client executing on a remote machine.

6.2. Experimental Results

In this section, we present some results of our study. The specific cases we chose to present capture the main traits of the wide range of results we obtained by varying the data, the query workloads, and system parameters.

On-line Tuning for Stable Workloads. In the first experiment, we compare the performance of COLT to OFFLINE for a workload with a fixed query distribution. This experiment has a duration of 500 queries. The workload implies a total of 18 relevant indices, many of which have high potential benefit. We select the space budget B so that it can fit 3 to 6 of these indices. This makes the task of on-line tuning nontrivial, as no materialized set is clearly optimal.

Figure 3 shows the relative performance of COLT and OFFLINE as time progresses. Each bar describes the sum of execution times for 50 queries in chronological order. The height of the grey region is the total execution time for the faster technique, either COLT or OFFLINE. Each black region indicates time spent by COLT in excess of time spent by OFFLINE. Similarly, the white regions indicate addi-

[2] A demonstration of our prototype appears in [18].

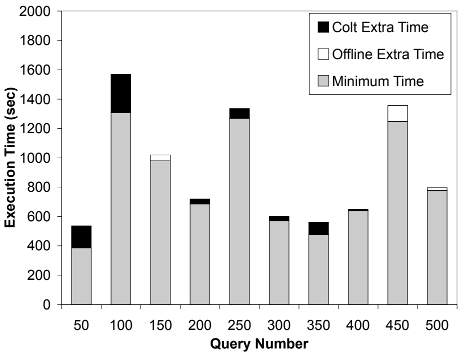

Figure 3. Stable workload.

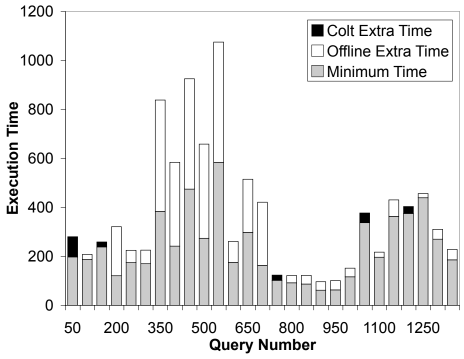

Figure 4. Shifting workload.

tional time spent by OFFLINE. During the first 100 queries, COLT has higher execution times as it monitors the query distribution, selects indices, and materializes them on disk. The overhead of index creation contributes significantly to the execution time for COLT during this period. After 100 queries, COLT has materialized the important indices for the workload, and the query execution time is essentially equal to OFFLINE with a negligible deviation of 1%. Clearly, this demonstrates that our on-line technique can achieve similar performance to the ideal off-line technique that has precise knowledge of the complete workload.

On-Line Tuning for Shifting Workloads. In the second experiment, we evaluate the performance of COLT on a shifting workload. We form a workload consisting of four distinct phases, each phase comprising 300 queries from a different query distribution. A particular phase focuses on specific attributes with different degrees of selectivity, and essentially implies a specific index set that is optimal for query evaluation. To make our experiment more realistic, we have tuned the distributions so that there is some overlap among the optimal index sets. Moreover, the transitions between phases occur gradually over 50 queries, implying a total of 1350 queries for the workload. The disk budget and total number of relevant indices are the same as the previous experiment.

Figure 4 shows the total query execution time under COLT and OFFLINE as the workload progresses. (We use the same illustration format as Figure 3.) The results clearly show that COLT outperforms the OFFLINE technique for the majority of queries in the workload. Being on-line, COLT can detect the different phases of query distribution and fine-tune the physical configuration accordingly; OFFLINE, on the other hand, selects an index set that is good *on the average* (i.e., for the complete workload) and thus misses significant opportunities for fine-tuning. This is evident in the second phase of the workload (queries 350-650), where the total query execution time under COLT is 49% shorter than OFFLINE – clearly, a significant reduction. Over the complete workload, on-line tuning results in 33% reduction in total execution time compared to the ideal off-line technique. We note that we have observed similar benefits in multi-user settings, where the shifting workload is generated by multiple concurrent clients. Overall, our results demonstrate the potential of on-line approaches in the design of self-tuning systems.

At this point, it is interesting to examine the overhead of self-tuning as the workload moves through the different phases. As described in Section 3, this overhead stems from the on-line maintenance of statistics and in particular from the use of additional what-if calls to the optimizer. Figure 5 charts the number of what-if calls invoked by COLT over each epoch during this experiment. Recall that we have a maximum number of 20 what-if calls for each epoch of 10 queries. The chart has four discernable peaks that coincide with the transitions to new query distributions. Apart from these peaks, COLT uses less than half of its budget in each epoch. This behavior matches the goals of our framework: COLT intensifies profiling when a shift is detected, and lowers the overhead when the workload is stable and the system is well tuned. We also observe that the workload queries have a significant number of relevant indices, but COLT judiciously profiles only 11% of these indices. Overall, this results in a very efficient use of the what-if optimizer.

Effect of Noise. Conceptually, we consider noise to be any query that does not reflect the dominant traits of the current query distribution. COLT is faced with a challenging task in this case, as it has to determine if few such queries constitute noise (and thus should be ignored), or whether they signal a new trend in the workload.

We consider a worst-case scenario for COLT and assume that noise queries occur in concentrated bursts. Depending on the length of a burst, an on-line tuning system may mistake it for a shift in the workload and change the configuration. We generate the test workload by starting with a fixed distribution $\mathcal{Q}_1$ and injecting bursts of queries from a distribution $\mathcal{Q}_2$. Each workload consists of at least 500 queries and contains at least 2 injections. (In all cases, the queries from $\mathcal{Q}_2$ represent 20% of the total workload.) We ensure

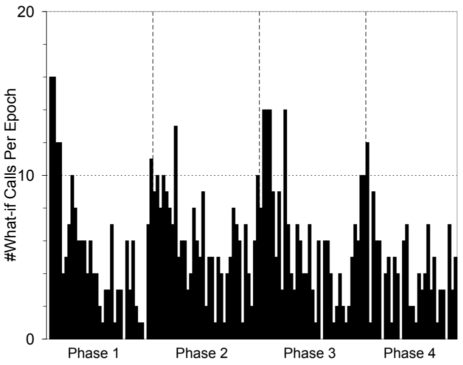

Figure 5. Overhead

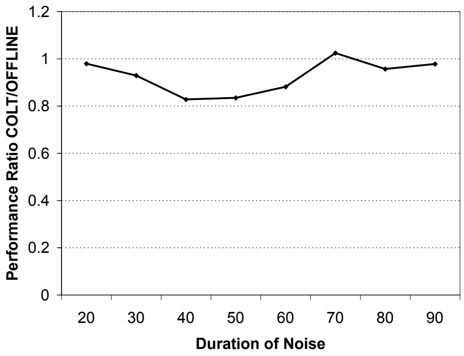

Figure 6. COLT **with noisy workload.**

that the optimal index sets for Q_1 and Q_2 are disjoint, and we start each workload with 100 queries from Q_1 in order to allow the system to stabilize before the noise. We vary the length of bursts from 20 to 80 queries. As our evaluation metric, we use the ratio of total query execution time under COLT to total query execution time under OFFLINE, where off-line tuning is performed solely on Q_1, i.e., it completely ignores noise.

Figure 6 shows the ratio of the execution time of COLT to the execution time of OFFLINE as a function of burst length. We do not include the first 100 queries in our measurements, because we want to isolate the queries that show the effects of noise. Our results indicate that COLT is resilient to short bursts of noise (up to 20 queries) and effectively ignores distribution Q_2. For long-lived injections (more than 70 queries), COLT materializes the index set for Q_2 relatively early and thus improves query performance considerably for several of the noise queries. Overall, COLT provides performance equivalent to OFFLINE for these ranges of bursts. Conversely, there is a small range of burst lengths (30-60 queries) where COLT becomes less effective than OFFLINE. For this range, the indices for Q_2 are materialized shortly after the burst begins, but they are not used extensively as the workload shifts quickly back to Q_1. Still, we observe that the average loss in performance is 18% compared to an ideal off-line technique having precise knowledge of the workload and noise.

As we describe in the full version of this paper, the worst burst length is correlated with the specifics of our forecasting model that predicts the future benefits of indices. In particular, our model uses a window of past measurements, which in this case coincides with the burst of noise queries and thus misjudges the utility of noise indices. It may be possible for the system to tune the length of this window if materialized indices are dropped too quickly. We plan to explore this extension in our future work.

7. Conclusions

This paper introduces COLT, a novel self-tuning framework that continuously monitors the incoming queries and adjusts the system configuration in order to maximize query performance. COLT minimizes the overhead of on-line tuning by carefully allocating profiling resources to the most promising candidate configurations, and by self-regulating its overhead. Our experiments validate the effectiveness of our approach. We show that COLT performs as well as off-line tuning for stable workloads and significantly outperforms off-line tuning for evolving workloads. Furthermore, COLT rapidly adapts to shifts of the query load, while being resilient to noise.

References

[1] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[2] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 683–694, New York, NY, USA, 2006. ACM Press.

[3] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 156–165, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[4] N. Bruno and S. Chaudhuri. Automatic physical database tuning: a relaxation-based approach. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 227–238, New York, NY, USA, 2005. ACM Press.

[5] N. Bruno and S. Chaudhuri. To tune or not to tune?: a lightweight physical design alerter. In *VLDB '06: Proceedings of the 32nd international conference on Very large data bases*, pages 499–510. VLDB Endowment, 2006.

[6] S. Chaudhuri, M. Datar, and V. Narasayya. Index selection for databases: A hardness study and a principled heuristic solution. *IEEE Transactions on Knowledge and Data Engineering*, 16(11):1313–1323, 2004.

[7] S. Chaudhuri and V. Narasayya. AutoAdmin what-if index analysis utility. In *SIGMOD '98: Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 367–378, New York, NY, USA, 1998. ACM Press.

[8] S. Chaudhuri and V. R. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 146–155, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[9] M. P. Consens, D. Barbosa, A. Teisanu, and L. Mignet. Goals and benchmarks for autonomic configuration recommenders. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 239–250, New York, NY, USA, 2005. ACM Press.

[10] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 330–341, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[11] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index selection for OLAP. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 208–219, Washington, DC, USA, 1997. IEEE Computer Society.

[12] M. Hammer and A. Chan. Index selection in a self-adaptive data base management system. In *SIGMOD '76: Proceedings of the 1976 ACM SIGMOD international conference on Management of data*, pages 1–8, New York, NY, USA, 1976. ACM Press.

[13] D. Kossmann, M. J. Franklin, G. Drasch, and W. Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Trans. Database Syst.*, 25(4):517–558, 2000.

[14] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed web sites. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 191–200, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[15] M. Palmer and S. B. Zdonik. Fido: A cache that learns to fetch. In *VLDB '91: Proceedings of the 17th International Conference on Very Large Data Bases*, pages 255–264, San Francisco, CA, USA, 1991. Morgan Kaufmann Publishers Inc.

[16] The PostgreSQL Database System `http://www.postgresql.org`.

[17] K.-U. Sattler, I. Geist, and E. Schallehn. QUIET: Continuous query-driven index tuning. In *VLDB '03: Proceedings of the 29th International Conference on Very Large Data Bases*, pages 1129–1132, San Francisco, CA, USA, 2003. Morgan Kaufmann Publishers Inc.

[18] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. COLT: continuous on-line tuning. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 793–795, New York, NY, USA, 2006. ACM Press.

[19] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis. Online database tuning. Technical Report UCSC-CRL-06-07, UC Santa Cruz, 2006.

[20] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD '79: Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pages 23–34, New York, NY, USA, 1979. ACM Press.

[21] Student. The probable error of a mean. *Biometrika*, 6(1):1–25, 1908.

[22] G. Valentin, M. Zuliani, D. C. Zilio, G. M. Lohman, and A. Skelley. DB2 advisor: An optimizer smart enough to recommend its own indexes. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, pages 101–110, Washington, DC, USA, 2000. IEEE Computer Society.

[23] D. C. Zilio, J. Rao, S. Lightstone, G. M. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. DB2 design advisor: Integrated automatic physical database design. In *VLDB '04: Proceedings of the 30th International Conference on Very Large Data Bases*, pages 1087–1097, San Francisco, CA, USA, 2004. Morgan Kaufmann Publishers Inc.