A Benchmark For Online Index Selection

Karl Schnaitter [#1], Neoklis Polyzotis [#2]

[#]*Computer Science Department, Univ. of California Santa Cruz*
Santa Cruz, CA, USA
[1]karlsch@soe.ucsc.edu
[2]alkis@soe.ucsc.edu

***Abstract*— Online approaches to physical design tuning have received considerable attention in the recent literature, with a focus on the problem of online index selection. However, it is difficult to draw conclusions on the relative merits of the proposed techniques, as they have been evaluated in isolation using different methodologies. In this paper, we make two concrete contributions to address this issue. First, we propose a benchmark for evaluating the performance of an online tuning algorithm in a principled fashion. Second, using the benchmark, we present a comparison of two representative online tuning algorithms that are implemented in the same database system. The results provide interesting insights on the behavior of these algorithms and validate the usefulness of the proposed benchmark.**

I. Introduction

Recent studies [1], [2], [3] have advocated an online approach to the problem of physical design tuning. In a nutshell, an online tuning algorithm monitors and analyzes continuously the current workload, and changes automatically the physical design to maximize the efficiency of query processing. Contrary therefore to off-line tuning [4], [5], [6], where the system is tuned based on a representative workload, the main assumption is that the workload is volatile and requires the system to be retuned repeatedly.

Current research efforts have focused on the variant of the problem that deals with online index selection. The proposed tuning algorithms [1], [2], [3] cover a wide gamut of features with respect to the type of indices that can be materialized, the performance guarantees enabled by the algorithm, and the complexity of the workloads that can be handled. However, each algorithm has been evaluated in isolation in the respective study, using a different methodology and execution environment. This makes it difficult to draw conclusions on the relative merits of the proposed techniques. Clearly, *it is desirable to evaluate empirically the proposed tuning algorithms in the same execution environment using a common methodology*.

A natural question is whether we can reuse the methodology of a previous study as the basis for the comparison. Unfortunately, the reported experimental methodologies are specific to each study and fail to exercise several important aspects of the proposed algorithms. For instance, the experimental study presented in [3] does not employ workloads with update statements, which can affect index performance and thus the decisions of the tuning algorithm. As another example, the methodology in [1] employs workloads of limited volatility, which does not allow stress testing the performance of online tuning. Overall, *it is desirable to design a principled experimental methodology for benchmarking the performance of online tuning algorithms*.

In this paper, we make concrete contributions in the previously outlined directions. First, we propose a principled benchmarking methodology for evaluating the performance of online tuning algorithms. The proposed benchmark consists of several workload suites, designed to exercise specific aspects of a tuning algorithm. The workloads are constructed using a general methodology that can be used to generate additional interesting suites. Using the benchmark, we then present an empirical evaluation of two representative online tuning algorithms that we have integrated in the PostgreSQL DBMS. The results lead to interesting insights about the relative merits of the two algorithms, and are thus of interest to both database researchers and practitioners who wish to employ the existing algorithms. In that sense, the results also validate the usefulness of the benchmark as a platform of comparison. To the best of our knowledge, this is the first study to examine different online tuning algorithms using the same database system and a common experimental methodology.

II. Online Index Selection: Problem Statement

We define a configuration as a set of indices that can be used in the physical schema. We use $\mathcal{S}$ to denote the space of possible configurations, and note that in practice $\mathcal{S}$ contains all sets of indexes that can be defined over the existing tables, whose required storage is below some fixed limit. We use $cost(q, C)$ for the cost of evaluating query q under configuration C. There is also a cost $d(C, C')$ involved with changing between two configurations $C, C' \in \mathcal{S}$.

We formulate the problem of online index selection in terms of online optimization. In this setting, the problem input provides a sequence of queries $q_1, q_2, \ldots, q_n$ to be evaluated in order. The job of online index selection is to choose for each q_i a configuration $C_i \in \mathcal{S}$ to process the query, while minimizing the combined cost of queries and configuration changes. In other words, the objective of an online tuning algorithm is to choose configurations that minimize

$$OBJ = \sum_{i=1}^{n} d(C_{i-1}, C_i) + cost(q_i, C_i)$$

where C_0 denotes the initial configuration. Furthermore, the online algorithm must select each configuration C_i using knowledge only from the preceding queries $q_1, \ldots, q_{i-1}$, i.e.,

without knowing the queries that will be executed in the future. This is precisely what gives the problem its online nature.

OBJ has been used in studies of metrical task systems [7], [8] and also in more closely related work in online tuning [1]. It is a natural choice to describe the problem, as it accounts for the main costs of query processing and index creation that are affected by online index selection. It is also straightforward to compute using standard cost models, making it a useful yardstick for measuring the performance of a self-tuning system. On the other hand, it does not model all practical issues involved with online tuning. For instance, observe that our statement of the problem assumes that the configuration C_i is always installed before query q_i is evaluated. This may not be feasible in practice because there may not be enough time between q_{i-1} and q_i to modify the physical schema, unless the evaluation of q_i is delayed by the system. It is more realistic to change the index configuration concurrently while queries are answered, or wait for a period of reduced system load.

The OBJ metric also does not reflect the overhead work required to analyze the observed queries $q_1, \ldots, q_{i-1}$ in order to choose each C_i. One significant source of this overhead can be the use of *what-if* optimization [6], [9] to estimate $cost(q, C)$ for a candidate configuration C. The what-if optimizer simulates the presence of C during query optimization to obtain the hypothetical cost, which is used in turn to gauge the benefit of materializing C in the physical schema. Although this is an accurate measure of query cost, each use of the what-if optimizer can take significant time, akin to normal query optimization. It is thus important for a tuning algorithm to limit the use of what-if optimization, or evaluate $cost(q, C)$ more efficiently.

III. Existing Approaches to Online Index Selection

The earliest study of online index selection known to us is over three decades old [10]. There has recently been a renewed interest in this problem, with three proposed algorithms in the past two years: the work of Bruno and Chaudhuri [1] (henceforth referred to as BC), COLT [3], and the "soft index" system of Sattler et al. [2] (which we will call SOFT). Below, we discuss the salient features of these algorithms and classify their designs along different dimensions that affect critically the performance of online index selection. These dimensions also become significant in the development of the benchmark that we describe in the next section.

Decision Logic: We begin with the most important dimension that concerns the underlying logic driving the decisions of the algorithm. The approaches that have been studied can be classified as either "predictive" or "retrospective". A *predictive* approach makes predictions about future queries, and chooses indices that have the most benefit with respect to those predictions. Both COLT and SOFT are predictive algorithms, but employ different forecasting models to make predictions about the future benefit of indices. On the other hand, the strategy of BC is better described as *retrospective*: the idea is to analyze the history of queries in hindsight to determine what indices would have been optimal in the past, then schedule these indices to be materialized next. Overall, we expect predictive approaches to perform well when the workload fits the assumptions of the forecasting model, whereas retrospective approaches guard against adversarial workloads by only creating indices that would be created earlier by an optimal strategy.

Candidate Indices: All of the algorithms in our discussion initialize their search of the configuration space by generating interesting index candidates for individual queries. COLT only considers single-column indices for attributes that appear in selection predicates. Both BC and SOFT allow for indices with multiple columns, by looking at all clauses of a query. However, the selection criteria for the two algorithms differ: SOFT bases its choices on the syntax of the query, whereas BC chooses candidates based on the optimized physical plan.

Index Interactions: Several previous works [11], [12] have observed that a major challenge in index selection stems from *index interactions*, i.e., cases where the benefit of an index is affected by the presence of other indices. COLT implicitly accounts for some interactions involving materialized indices, because these indices are considered to be available when using the what-if optimizer. SOFT and BC take a more proactive approach to model index interaction using specific formulas to manipulate the measured benefits of candidate indices.

Cost Estimation: An important component of any tuning algorithm is the mechanism to estimate the cost of queries using hypothetical indices (i.e., the $cost(q, C)$ metric for a hypothetical configuration C). This component typically entails what-if optimization and thus affects directly the overhead of online tuning. SOFT uses a what-if optimizer for all estimations of $cost(q, C)$. COLT also makes use of a what-if optimizer, but tries to reduce overhead by limiting the frequency of what-if calls and reusing the $cost(q, C)$ values for different queries. The BC algorithm takes a very different approach: It computes an *upper bound* on $cost(q, C)$ by making minor transformations to the optimal plan of q under the materialized configuration. This avoids the cost of what-if optimization since the optimal plan is a free byproduct of processing q.

IV. Benchmark Description

At a high level, the proposed benchmark models a hosting scenario where the same system hosts several separate databases, and the workload may shift its focus to different subsets of the databases over time. In what follows, we discuss the specifics of the benchmark, starting with the environment in which the online tuning algorithms must operate and the metrics used to quantify their overall performance. We then describe the data sets employed in our benchmark and the methodology for generating shifting workloads. Finally, we define the specific test cases that comprise the benchmark. These cases are organized into four *workload suites* that exercise different aspects of an online tuning algorithm.

A. Operating Environment

As mentioned above, our benchmark uses an environment in which several databases reside on a single server. More specifically, we assume that the data is stored in a traditional relational database system. It is preferable for the system to employ state-of-the-art query processing techniques, but this leaves some ambiguity, since existing DBMSs are not all designed with the same features. This is an important point because online tuning algorithms typically depend on the design of the database system. For example, COLT and SOFT require the optimizer to provide a what-if interface, and BC assumes that queries are processed by tree-structured physical plans. Rather than fix a particular set of DBMS features, we require the system to implement the features that are assumed by the online tuning algorithms under evaluation, provided that the assumptions of different algorithms do not conflict. This requirement is not ideal, since it causes the benchmark specification to depend on the algorithms being tested, but it avoids the greater danger of choosing system features that are biased toward one index selection algorithm. Section V details the the database features involved in our benchmark implementation.

Another important aspect of the environment is the maximum storage allowed for indices created by the online tuning algorithm, which we refer to as the *storage budget*. A larger storage budget intuitively makes the problem easier since it allows a large number of indices to be materialized without needing to choose indices to evict. On the other hand, the budget should be large enough for the tuning algorithm to materialize an index configuration with significant benefit. Based on these observations, we compute the physical storage required by the tables in each database, and set the storage budget to the mean of the database sizes. This results in a moderate storage budget that is interesting for index selection.

B. Benchmark Metrics

We employ two metrics to measure the performance of a tuning algorithm for a specific workload. The first metric is the OBJ metric defined in Section II that captures the total work to run the workload statements under the current configuration and to materialize indices. The work for each of these tasks is estimated using the cost model of the query optimizer, so as to measure performance under the same "world model" that the online tuner consults to make its decisions. As argued in a previous study [13], this metric is robust in the sense that it is not affected by any mismatch between the statistics of the optimizer and the actual execution environment.

As a complementary performance metric, we measure the total wall clock time to complete the workload. To ensure determinism and some statistical stability in measurements, we replay the workload using a single client with a cold cache. The measured time includes two components that are not measured by the ideal metric, namely, the overhead of online tuning (e.g., what-if optimization) and the slow-down caused by asynchronous index materializations. Another difference is that the benefit brought by an index appears only after the index is materialized, whereas the ideal metric assumes that the index is available to use for the query immediately following the decision to create the index. The use of both metrics thus provides a well-rounded approach to evaluating a tuning algorithm.

For both metrics, we report the improvement brought by online tuning compared to a baseline system configuration that does not contain any indices (except perhaps for any primary or foreign key indices that are created automatically). We choose this "naked" baseline in order to gauge the maximum benefit of a particular online tuning algorithm. More concretely, assuming that X_{ot} and X_b measure the cost of the online tuning system and the baseline system respectively using one of the aforementioned metrics, we measure the performance of online tuning with the ratio $\rho = (X_b - X_{ot})/X_b$. The sign of ρ indicates whether online tuning improves ($\rho > 0$) or degrades ($\rho < 0$) system performance compared to the baseline. The absolute value $|\rho|$ denotes the percentage of improvement or degradation.

C. Databases

We use the following data sets as the distinct databases of the benchmark: TPC-H, TPC-C, and TPC-E from the TPC benchmarking suite [1], and the real-life NREF data set [2]. (Note that TPC-H and NREF have been used in a previous benchmarking study on physical design tuning [14].)

The selection of these specific data sets is not crucial for the benchmark. Indeed, it is possible to apply the overall methodology to a different choice of databases. Any selection, however, should contain sufficiently different data sets, so that we can create a diverse workload by shifting from one database to another. On the other hand, the data sets should be roughly equal in size, so that no database becomes too cheap (conversely, too expensive) to process. To model an interesting setting, we require that the total size of the databases exceeds the main memory capacity of the hardware on which the benchmark executes.

Our choice of databases make use of some synthetic data with attribute values generated from a uniform distribution. This does not necessarily match the characteristics of real-life data sets, but we work around this issue in our workload generation methodology: the main idea is to directly control the selectivity of predicates, which does not rely on the data following any particular distribution (more details below).

D. Workload Generation

We present next a systematic methodology for generating shifting workloads over the set of loaded databases.

Query-Only Workloads: We define a workload *phase* as a sequence of queries drawn from a specific distribution of database popularity. The distribution is described by a vector of positive weights $(w_1, \ldots, w_m)$, where $m = 4$ is the number of databases and $\sum w_i = 1$. The weight w_i indicates the probability of observing a query on database i, $1 \leq i \leq m$.

[1] http://www.tpc.org

[2] http://pir.georgetown.edu/pirwww/dbinfo/nref.shtml

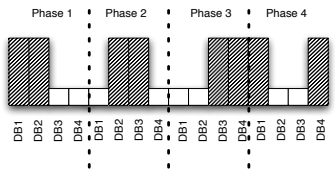

Fig. 1. Weight vectors for the phases in a query workload. Shaded bars denote the focus.

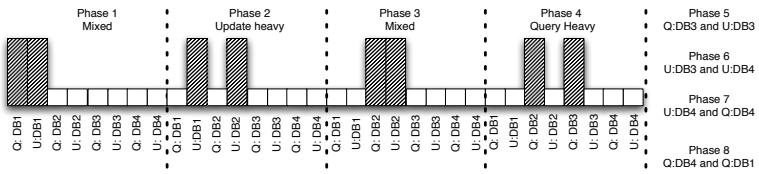

Fig. 2. Weight vectors for the phases in an update workload. Shaded bars denote the focus of the phase. The notation "Q: DB1" denotes queries on database DB1, whereas "U: DB1" denotes updates.

Thus, the next query for the phase is generated by first sampling a database index and then generating a random query for the specific database. (We discuss query generation below.) A workload is created by concatenating several phases of the same number of queries. To avoid abrupt and easily detected shifts, the workload shifts gradually from one phase to the next, using a transitional period with a "mixed" distribution. We do the shift during the last 10% and first 10% of the queries in each phase, with a linear transition between the two query distributions. For example, if there are 100 queries per phase, then queries 91–110 will be drawn from a mix of the query distributions for the first two phases.

We use three possible templates to generate random queries that reflect different levels of workload complexity. In the description that follows, a low selectivity predicate refers to a range predicate with a selectivity factor randomly distributed in $(0, 0.01]$. A high selectivity predicate is defined analogously for the interval $(0.01, 1]$.

– A *simple query* computes a COUNT(*) aggregate over a single table, applying a single low selectivity predicate. This template represents the simplest case for online tuning: an index on the predicated attribute always brings a large benefit, as it covers the query and has to retrieve fewer than 1% of its entries; and, all queries employ a single predicate, which excludes any index interaction in query evaluation. These properties simplify considerably the bookkeeping that needs to be performed by an online tuning algorithm.

– A *moderate query* is similar to a simple query but applies a conjunction of low and high selectivity predicates on several attributes. This query class is more challenging for online tuning: high selectivity predicates imply that not all indices are beneficial; the covering index for a query may be multi-column; and, physical plans can employ index intersection, which implies that indexes may interact. Clearly, these features may complicate the bookkeeping of a tuning algorithm.

– A *complex query* may involve several relations linked through key/foreign-key equi-joins. It also applies several selection predicates similar to a moderate query. This template introduces an additional challenge for an online tuning algorithm, as the query can now benefit from indexes on the join predicates. Moreover, index interactions become more involved, as join indices may reduce the benefit of indices on predicated attributes (and vice versa).

We employ a 50/50 mix of low and high selectivity predicates for moderate and complex queries. For all query templates, the tables participating in each query are chosen at random, with a selection probability that is proportional to the corresponding tuple cardinality. This method mirrors the common assumption that query distribution follows data distribution. Given a table, selection predicates can be placed only on a subset of its non-key attributes which we term the *predicated attribute-set*. The size of each predicated attribute-set is bounded by a benchmark parameter that essentially controls the degree of workload variability. More concretely, given a value for this bound, say k, the predicated attribute-set contains the k attributes with the largest active domains. The rationale is that these attributes are the most "interesting" in terms of predicate generation. Thus, a low bound implies that fewer attributes carry predicates, which in turn results in fewer candidate indices that can benefit more queries. In a sense, the tuning problem becomes easier in this scenario of low variability. Conversely, tuning becomes more challenging as the number of predicated attributes increases, since there are more candidate indices and their benefit may not be as clear. As a convention, we assume that the predicated attribute-set contains all the non-key attributes of each table if the bound is equal to infinity.

An alternative approach to random queries would be to use standardized query sets that come with the chosen databases. As an example, for the TPC-H database, we could draw at random from the predefined TPC-H queries. However, we still need to define a query generator for databases that do not come with a standardized query set, e.g., NREF. It is thus meaningful to use the same generator for all databases to ensure uniformity. Moreover, it is difficult to create interesting scenarios for online tuning using standardized workloads, as the latter are not designed for this purpose. For instance, it is challenging to create a diverse workload using just the 22 queries of the TPC-H workload.

Workloads with queries and updates: We extend the previous methodology to the generation of workloads with queries and updates. The distribution in each phase is now described with a vector $(w_1, \ldots, w_{2m})$, where w_{2i} is the probability of observing a query on database i, and w_{2i+1} is the probability of observing an update on the same database, $1 \leq i \leq m$. The stitching of several phases is done in a similar fashion as for queries.

We generate a random update statement by building an UPDATE SQL command on a randomly chosen attribute. The generated commands follow two possible templates, namely, Simple and Complex updates. Simple updates employ a single

Suite	Number of Phases	Phase Length	Distribution of DB popularity	Query Template	Update Template	Maximum size of predicated attribute-set
C1	4	50 100 200 400	80-20	Moderate	-	∞
C2	4	300	80-20	Simple Moderate Complex	-	∞
C3	4	300	80-20	Moderate	Simple Complex	∞
C4	1	500	Uniform	Moderate	-	∞
C5	4	300	80-20	Moderate	-	1 2 3 4

TABLE I

WORKLOAD GENERATION PARAMETERS FOR THE TEST SUITES.

low selectivity predicate in the WHERE clause of the statement, whereas complex updates have a conjunction of low and high selectivity predicates (similar to the moderate and complex query templates). In both templates, the SET clause applies a small randomized modification to one attribute so that the expected change over multiple update statements is equal to 0. This approach ensures that the update statements do not modify the data distribution significantly.

E. Workload Suites

The core of the benchmark consists of four workload suites that exercise specific features of an online tuning algorithm. Each workload suite uses a variety of parameter settings for our workload generation methodology. For ease of reference, Table I summarizes the parameter settings for each suite.

C1: Reflex: The first suite examines the performance of online tuning for shifting workloads of varying phase length. This parameter essentially determines the speed at which the query distribution changes, and thus aims to test the reaction time of an online tuning algorithm.

A workload contains queries in the moderate class, which involve multi-column covering indices that may interact. There are $m = 4$ phases, whose weight vectors are shown in Figure 1. Each phase concentrates most of the accesses on two databases using an 80-20 rule, i.e., 80% of the queries are spread uniformly across two databases and the remaining 20% on the other two databases. The focus of the workload shifts to two different databases with each phase, ensuring that consecutive phases overlap on exactly one database. This implies that consecutive phases have overlapping sets of useful indices, which creates an interesting case for online tuning.

The suite comprises four workloads with phase length 50, 100, 200, and 400. An aggressive tuning algorithm is likely to observe good performance across the board, even when transitions are short-lived. A conservative algorithm is likely to view short-lived transitions as noise, and show gains only for larger phase sizes.

C2: Queries: The second test case examines the performance of online tuning as we vary the query complexity of the workload. This parameter affects the level of difficulty for performing automatic tuning.

A workload consists of m phases, with the same weight vectors as in the previous experiment. Thus, we have a gradually shifting workload with overlapping phases. The phase length is set to 300 queries, under the assumption that this is long enough for a reasonable algorithm to adapt to each phase.

We generate three workloads consisting of simple, moderate, and complex queries respectively. The simple workload is expected to yield the highest gain for any tuning algorithm. The moderate workload exercises the ability of the algorithm to handle multi-column indices and index interaction. Finally, the complex workload introduces join indices to the mix.

C3: Updates: The third suite examines the performance of a tuning algorithm for workloads of queries and updates. The goal is to test the ability of the algorithm to capture both the benefit and the maintenance overhead of indices.

A workload in this suite consists of $2m$ phases, following the ordering shown in Figure 2. Each phase applies again an 80-20 rule to determine the weight vector. The ordering of phases creates the following types of phases: mixed, where the same database is both queried and updated; query-heavy, where most of the statements are queries over two different databases; and, update-heavy, where most of the statements are updates over two different databases. The idea is to alternate between phases with a different effect on online tuning. For instance, a query-heavy phase is likely to benefit from the materialization of indices, whereas an update-heavy phase is likely to incur a high maintenance cost if any indices are materialized. Mixed phases fall somewhere in the middle, since the same database is both queried and updated.

We generate two workloads with simple and complex updates respectively. In both cases, the queries are of moderate complexity. Clearly, simple updates involve less complicated index interactions and hence the bookkeeping is expected to be simpler. Conversely, we expect online tuning to be more difficult for updates of moderate complexity.

C4: Convergence: This suite examines online tuning with a stable workload that does not contain any shifts, i.e., the

whole workload consists of a single phase. The expectation is that any online tuning algorithm will converge to a configuration that will remain unchanged by the end of the workload. Thus, it is interesting to identify the point in the workload where the configuration "freezes".

The suite consists of a single workload with a single phase of 500 moderate queries. The phase employs a uniform weight vector that places equal importance to all databases, i.e., $w_i = 1/m$. This creates a setting where the choice of optimal indices is less obvious, compared to a phase that focuses on a specific set of databases.

C5: Variability: The last suite examines the performance of online tuning as we expand the set of attributes that carry selection predicates. The goal is to evaluate the effect of this type of workload variability on a tuning algorithm.

The workloads in this suite are identical to suite C2, except that we vary the maximum size of the predicated attribute-sets as 1,2,3, and 4. Recall that a small maximum size implies that fewer attributes can receive selection predicates, which makes the workload less variable. We expect to observe a direct correlation between the variability of the workload and the effectiveness of an online tuning algorithm, but it is also interesting to observe the relative difference in performance for the different values of the varied parameter.

V. Benchmark Implementation

To validate the proposed benchmark, we use it to compare experimentally two online tuning algorithms, namely, COLT and BC. These algorithms form an interesting selection because they differ significantly (Section III) and most importantly because they represent the predictive and retrospective approaches to online tuning.

A. Software Platform

We implemented both algorithms inside PostgreSQL. We chose PostgreSQL as our testing platform since it is a mature, robust system and the source code is freely available.

Our implementation of BC inside PostgreSQL had to address some technical issues, due to certain assumptions behind the design of the algorithm. First, the index selection of BC is guided in part by information that is generally available from a top-down query optimizer. Since PostgreSQL uses a bottom-up optimizer, our implementation needed to simulate this information as an additional step in the algorithm. Second, BC tries to choose indexes that cover all attributes required by a query, in order to avoid accessing the base table for additional attributes. In PostgreSQL, indices are "lossy" in the sense that they may return recently deleted tuples, hence all index scans require access to the base table in order to filter out false matches. Therefore, we extended the indexing system of PostgreSQL to support lossless indices. Overall, we strived to implement BC so that its performance in PostgreSQL is correlated to its performance in a system with native support for top-down optimization and lossless indices.

Our implementation of COLT followed the original algorithm very closely. Our only change was an extension of the forecasting model to account for index maintenance costs associated with updates. We allowed COLT to use the lossless indices that are required by BC, since COLT is not geared toward any particular index format, and the use of lossless indices allows us to judge COLT and BC on equal grounds. Finally, we note that we extended the PostgreSQL optimizer to allow COLT to perform what-if optimization.

B. Benchmark Setup

We used a dedicated Linux server to run the DBMS (2x2.8GHz Pentium 4, 4GB RAM, Kernel 2.6.18-8.el5 with SMP support, 7200RPM SATA disk).

PostgreSQL reported a total size of 4GB after loading all data and building indices for primary keys. Of this, 2.9GB was attributed to the base table storage. As specified by the benchmark, we set the index storage budget to the mean of the four database sizes, which in our case was about 750MB. To create an interesting execution environment, we artificially limited the available RAM to 1GB and set the DBMS buffer pool to 64MB.[3] We note that we kept a small data scale to ensure timely completion of the experiments.

The workloads were replayed on a separate client machine that connected to the database server. Time measurements were taken at the server using the `gettimeofday` system call.

C. Results

We now discuss the results of executing the workload suites described in Section IV. Due to limited space, our presentation focuses on the OBJ metric, only reporting the wall-clock time when it is most interesting.

Algorithm Reactivity: Figure 3 shows the performance of COLT and BC on test suite C1, which uses workloads with varying phase lengths. The plotted performance metric is the improvement in the OBJ metric, taking all queries of the workload into account. It is clear that both algorithms perform better as the phase length increases, which is intuitive since longer phases allow for more opportunity to adapt to shifts in the workload. The worst case occurs with a phase length of 50, where both algorithms have very small benefit according to the OBJ metric.

The results with the wall clock time metric are qualitatively the same in most cases, but they can also reveal interesting details about the performance of the two algorithms. Recall that OBJ simulates a scenario where an index is available to use immediately after choosing it for the configuration. This is especially unrealistic when the phases are short, since the latency to build an index is significant with respect to the length of the period where the index is useful. The wall clock time metric gives a more realistic view of performance in this case, as the latency of index creation is captured in the metric. Indeed, this intuition is verified in Figure 4, which depicts the wall clock time for the shortest phase length $= 50$. The plot shows for each i the improvement in wall clock time for the

[3]We followed the practice advocated by PostgreSQL administrators to delegate cache management to the file system.

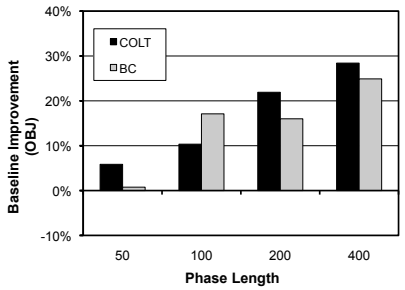

Fig. 3. Algorithm Reactivity. Benefit is measured for varying phase lengths using estimated costs.

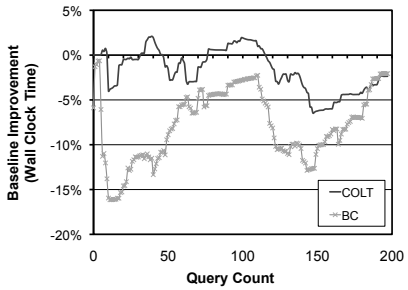

Fig. 4. Performance with a phase length of 50, according to wall clock time.

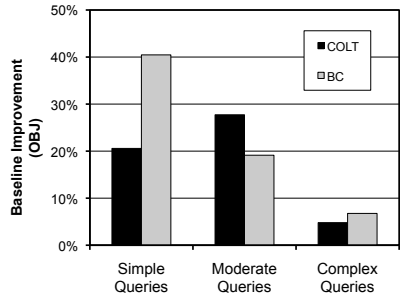

Fig. 5. Effect of Query Complexity. Benefit is measured using estimated costs.

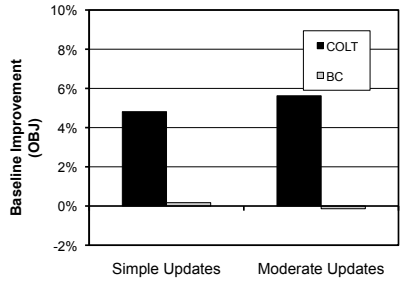

Fig. 6. Update workloads. Benefit is measured using estimated costs.

first i queries in the workload. The benefit of COLT dominates BC for most of the time, but the final measurement shows that both algorithms slightly degrade performance. This indicates that the OBJ metric may be too rough an approximation for a rapidly changing query distribution.

Comparing for other phases, we observe that BC has more benefit than COLT for a phase length of 100, but COLT seems to have more benefit than BC for workloads with longer phases. This matches our intuition discussed in Section III: a retrospective algorithm will have better stability in cases with a high degree of volatility in the workload, since it carefully creates indices only if they were optimal in the past. In contrast, a predictive approach tries to identify patterns in the workload, and performs better when the phases are longer and those patterns are clear.

The Effect of Query Complexity: Figure 5 shows the results from test suite C2, which explores the effect of complexity in the individual queries of the workload. Again, we plot the relative improvement in the OBJ metric. We observe that BC outperforms COLT for the simple workload, but the trend is reversed for moderate queries. This is surprising, since the moderate queries use several column from each table, and BC should be able to take advantage of multi-column indices. We further explore this point below, in our discussion of the Variability suite. We also observe that both techniques have the lowest performance for complex queries. It is not clear whether there is another algorithm that would have a better showing, but these queries provide an interesting challenge for the future development of tuning algorithms.

It is interesting to examine closer the behavior of the two algorithms on moderate queries. Essentially, COLT materializes several single-column indices which can benefit many queries in the workload. On the other hand, BC considers multi-column indices that cover at least one past query. These indices can yield significant benefits, as evidenced by the performance of BC, but they are more specific and also fewer of them can be materialized due to their size. These results indicate that a middle-ground approach may outperform both algorithms. The main idea would be to consider multi-column indices for materialization, without restricting the selection to wide (and large) covering indices.

Update Workloads: The results of our experiments with updates are shown in Figure 6. The chart shows that COLT has a small benefit on these workloads, around 5%. The BC algorithm, on the other hand, shows almost no net effect on performance. When analyzing the behavior of the algorithms, we found that BC had a tendency to create a much larger number of indices than COLT, by a factor of about 3. We have also observed this type of behavior in other workload suites. The reasons involve some detailed understanding of the algorithms, but the high level reason for this is that COLT requires indices to show their benefit frequently to take notice, whereas BC may create an index even if it's relevant queries are widely spread out.

In many cases, the more eager materialization is beneficial because the indices will eventually be useful enough to outweigh their materialization cost. In these workloads, however, an index may be beneficial for some phase, but then require

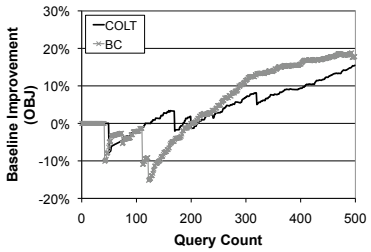

Fig. 7. Performance with static workload distribution, according to estimated cost.

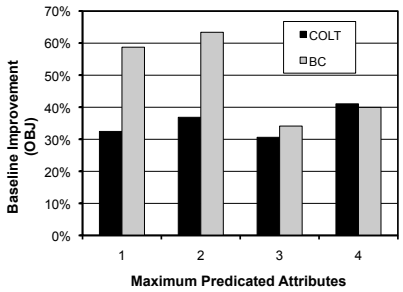

Fig. 8. Effect of the maximum number of predicated attributes per table. Benefit is measured using estimated costs.

high maintenance costs after a phase shift. This is a difficult issue for any tuning algorithm to handle, but the problem is slightly more amplified in the case of BC due to its larger materialized set.

Static Workload Distribution: We next show our results from the Convergence workload suite. Our results are shown in Figure 7, which shows the evolution of the OBJ metric over time. The first observation that we can make is that the net effects of the two algorithms are very similar, providing a benefit of about 15%. We can make more interesting observations from the shapes of the lines. Recall that this workload has a static query distribution, so the best strategy should intuitively create a good set of indices, and then stop changing the materialized set. We see that BC creates several indices in the early stages of the workload (signaled by downward jumps) and after about 120 queries, the benefit steadily increases because the set of indices has converged. COLT, on the other hand, creates indices more slowly, including an index that is created after more than 300 queries. The faster convergence of BC may be preferred in some contexts, since it implies a short period of instability followed by a long period of consistent performance. It is also advantageous to create indices early and use them for as many queries as possible—this is illustrated by the results, since BC has better overall performance than COLT after the first half of the workload.

Effect of Workload Variability: Our last suite shows the behavior of the algorithms as we modify the maximum size of the predicated attribute set. The performance according to the OBJ metric is shown in Figure 8. There is a noticeable difference between the bounds 1,2 and the bounds 3,4. In the former case, BC has much higher benefit than COLT, and in the latter case, the algorithms are roughly equal. To understand this switch, we examined the statistics of candidate indices within the BC algorithm. We found that the more variable workloads caused BC to create a larger number of index candidates. Since BC considers arbitrary multi-column indices, the number of possible candidates grows very quickly with the number of predicated attributes. When the bound on the predicated attribute set size was ≥ 3, the number of index candidates became too large to accurately track the benefits of each one, which led BC to underestimate their benefit and materialize them more slowly. This is not surprising, since the number of relevant multi-column indices grows so quickly. On the other hand, this workload suite provides an excellent stress test to determine how well an algorithm scales with the number of predicated attributes.

REFERENCES

[1] N. Bruno and S. Chaudhuri, "An online approach to physical design tuning," in *ICDE '07: Proceedings of the 23rd International Conference on Data Engineering*, 2007.

[2] K.-U. Sattler, M. Lühring, I. Geist, and E. Schallehn, "Autonomous management of soft indexes," in *SMDB '07: Proceedings of the 2nd International Workshop on Self-Managing Database Systems*, 2007.

[3] K. Schnaitter, S. Abiteboul, T. Milo, and N. Polyzotis, "On-line index selection for shifting workloads," in *2nd International Workshop on Self-Managing Database Systems (SMDB)*, 2007.

[4] S. Agrawal, S. Chaudhuri, and V.Narasayya, "Automated Selection of Materialized Views and Indexes for SQL Databases," in *International Conference on Very Large Databases*, 2000, pp. 496–505.

[5] N. Bruno and S. Chaudhuri, "Automatic Physical Database Tuning: A Relaxation-based Approach," in *ACM SIGMOD International Conference on Management of Data*, 2005.

[6] S. Finkelstein, M. Schkolnick, and P. Tiberio, "Physical database design for relational databases," *ACM Trans. Database Syst.*, vol. 13, no. 1, pp. 91–128, 1988.

[7] A. Borodin, N. Linial, and M. E. Saks, "An optimal on-line algorithm for metrical task system," *Journal of the ACM*, vol. 39, no. 4, pp. 745–763, 1992.

[8] W. R. Burley and S. Irani, "On algorithm design for metrical task systems," *Algorithmica*, vol. 18, no. 4, pp. 461–485, 1997.

[9] S. Chaudhuri and V. Narasayya, "Autoadmin "what-if" index analysis utility," *SIGMOD Rec.*, vol. 27, no. 2, pp. 367–378, 1998.

[10] M. Hammer and A. Chan, "Index selection in a self-adaptive data base management system," in *ACM SIGMOD International Conference on Management of Data*, 1976, pp. 1–8.

[11] S. Chaudhuri and V. R. Narasayya, "An efficient cost-driven index selection tool for microsoft sql server," in *International Conference on Very Large Databases*, 1997, pp. 146–155.

[12] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden, "Db2 design advisor: integrated automatic physical database design," in *International Conference on Very Large Databases*, 2004, pp. 1087–1097.

[13] N. Bruno, "A critical look at the tab benchmark for physical design tools," *SIGMOD Rec.*, vol. 36, no. 4, pp. 7–12, 2007.

[14] 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*, 2005, pp. 239–250.