Benchmarking Adaptive Indexing

Goetz Graefe[2], Stratos Idreos[1], Harumi Kuno[2], and Stefan Manegold[1]

[1] CWI Amsterdam
The Netherlands
first.last@cwi.nl
[2] Hewlett-Packard Laboratories
Palo Alto, CA
first.last@hp.com

Abstract. Ideally, realizing the best physical design for the current and all subsequent workloads would impact neither performance nor storage usage. In reality, workloads and datasets can change dramatically over time and index creation impacts the performance of concurrent user and system activity. We propose a framework that evaluates the key premise of adaptive indexing — a new indexing paradigm where index creation and re-organization take place automatically and incrementally, as a side-effect of query execution. We focus on how the incremental costs and benefits of dynamic reorganization are distributed across the workload's lifetime. We believe measuring the costs and utility of the stages of adaptation are relevant metrics for evaluating new query processing paradigms and comparing them to traditional approaches.

1 Introduction

Consider the task of selecting and constructing indexes for a database that contains hundreds of tables with tens of columns each; a horrendous task if assigned purely to a DB administrator. Figure 1 illustrates the various methods on how to reach an appropriate physical design.

The simplest approach (top row) loads data quickly without indexes, and then does a full table scan for every query. Traditional offline approaches (2nd row) invest the effort to create certain indexes, and then enjoy fast queries on those indexed columns. A third approach is based on online tuning and loads data quickly without indexes. It first observes the workload and then identifies and constructs the most promising indexes [2,12,13]. Unlike all of these methods, *adaptive indexing* (bottom row) creates and refines indexes automatically and incrementally, as a side effect of query execution [6,7,9,10,11].

Each approach is ideal for certain scenarios, depending on how much workload knowledge and idle time is available to invest in preparations, how much storage space and maintenance effort we can afford to spend, and, last but not least, workload composition. For example, if a workload is well-understood and extremely unlikely to change, then it might be most effective to create indexes up-front, as shown in Figure 1(2).

R. Nambiar and M. Poess (Eds.): TPCTC 2010, LNCS 6417, pp. 169–184, 2011.

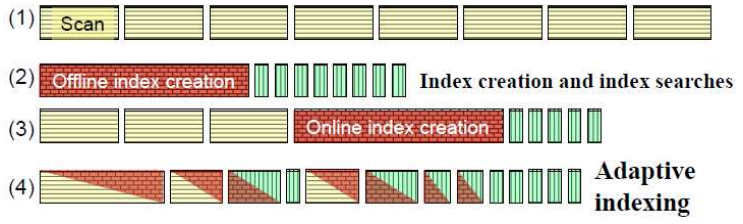

Fig. 1. Four approaches to indexing with regard to query processing

However, one can think of other scenarios with sudden, unpredictable, and radical workload changes. For example, usage of search engines follows current trends on news and human interest. A sudden event somewhere in the world is followed by millions of users searching for the same patterns for a limited amount of time. One cannot anticipate these events before-hand. Any effort to improve performance should have instant results, yet may be useful only during this workload peak, burdening the system afterwards unnecessarily with extra storage and maintenance effort. Adaptive indexing, Figure 1(2), can respond to workload changes automatically, yet without over-investing in short-lived trends.

Contributions. The first two approaches of Figure 1 have been extensively studied in the past, and recently an initial approach for benchmarking online selection (the third approach) has been proposed [14]. In this paper, we set forth a framework for evaluating the new approach of adaptive indexing so that we can properly and systematically compare adaptive indexing techniques as well as identify their benefits over past approaches in dynamic scenarios.

Dynamic Workloads. Adaptive indexing targets dynamic and unpredictable scenarios. For example, in a scientific database, researchers perform exploratory analysis to interpret the data and infer underlying patterns. Workload patterns continuously evolve with their exploration of the data [15]. With new scientific data arriving on a daily basis and with changing search patterns, there is little or no chance to settle for just one workload and create indexes only for that. By the time we have created indexes for one workload pattern, the focus may have already changed. In addition, blindly replicating data in such huge data sets is not appropriate given the already extensive use of storage resources of the ever-expanding data set.

With Knowledge and Time. Traditional approaches for physical design tuning are designed with a drastically different scenario in mind. Perfect up-front workload knowledge is assumed while workloads are assumed to be stable with enough idle time to accommodate traditional physical design. More recent approaches, i.e., soft indexes and online tuning [2,13], go one step further by providing a monitoring step that tries to understand the workload while the system is working and only then create the proper indexes. This deals with situations

where the workload is not known up-front but it also increases the delay of reaching good performance since queries during the monitoring period are not supported by indexes. Such approaches only make sense for scenarios where the time needed to spend in monitoring and the time needed to create the proper physical design are in proportion to the workload span and query performance without indexes is acceptable.

Continuous Physical Adaptation. Adaptive indexing, a very recent development in database architecture technology, addresses the above problems. Instead, of requiring monitoring and preparation steps in order to select and create useful indexes, index creation becomes an integral part of query processing via *adaptive database kernels*. The actual query execution operators and algorithms are responsible for changing the physical design. Essentially, indexes are built selectively, partially and incrementally as a side-effect of query execution. Physical changes do not happen after a query is processed. Instead, they happen while processing the query and are part of query execution.

Adaptive indexing can drastically change the way a database system operates. It also drastically changes the way we should evaluate query processing performance. Current benchmarks largely consider workload knowledge a given, while the index creation overhead is not taken into account as part of the processing costs. However, an adaptive indexing technique and relevant application scenarios need to be evaluated in an entirely different setting considering the complete costs as well as to take into account the various workload phases and how the system performance evolves. This changes the picture dramatically.

A New Benchmark. A recent benchmark proposal formalizes special requirements for online index selection [14]. Unlike established traditional benchmarks, new index structures are intended to be created on-the-fly, so this benchmark takes into account the cost of index creation.

In this paper, we outline a new benchmark specifically targeted for the evaluation of adaptive indexing implementations. As in online tuning, base data is loaded without incurring any time cost for index maintenance before the first query arrives. However, unlike the scenario considered by [14], index creation and maintenance efforts are integrated into query execution actions. Given this incremental, continuous and selective nature of adaptive indexing we need a very different evaluation method than does online index selection.

Breaking the Workload Span. How good or bad an adaptive indexing technique is for a particular workload depends on the overhead that incremental indexing adds to each query and how many queries benefit from the incremental indexing, versus the degree to which that query benefits from the efforts of prior queries. Thus, how an adaptive indexing system compares to a system without index support (in terms of fast loading) or to a system with full index support (in terms of fast queries) depends on how incremental physical design actions are *performed and scheduled* during the workload span. We believe that such new methods are required to evaluate how well query processing techniques serve workloads that are increasingly complex, dynamic, and possibly mixed.

Outline. The rest of this paper is organized as follows. Section 2 briefly discusses related work. Then, Section 3 describes the benchmark in detail while it also provides examples of adaptive indexing behavior and evaluation. Section 4 discusses partial reference implementations of the benchmark, and Section 5 proposes future work. Finally, Section 6 concludes the paper.

2 Related Work and Background

2.1 Classic and Online Indexing

Typically, indexes have been selected and refined by database administrators, possibly using physical design tools, in an offline process based on analyzing a known representative set of the workload [4]. Indexes are then created wholesale. State of the art research also suggests the usage of *alerters* [1,3], i.e., monitoring tools that alert the DBA on when the system should be re-tuned in order to refine a currently suboptimal physical design.

More recently online index creation approaches have been introduced [2,12,13]. They extend the above model for the cases where the workload is not known upfront. They add a monitoring step while the actual workload is being processed and an index is created automatically only once it is believed that it will pay off. Indexes are again created in one go and completely with the difference being that they are created in the background while the workload is actually running.

2.2 Adaptive Indexing

Here, we briefly sketch two adaptive indexing techniques we have recently introduced.

Database Cracking. Database cracking [10] combines some features of both automatic index selection and partial indexes to implement adaptive indexing. As shown in Figure 1(4), it reorganizes data within the query operators, integrating the re-organization effort into query execution. When a column is queried by a predicate for the first time, a new cracker index is initialized. As the column is used in the predicates of further queries, the cracker index is refined by range partitioning until sequentially searching a partition is faster than binary searching in the AVL tree guiding a search to the appropriate partition. The keys in a cracker index are partitioned into disjoint key ranges, but left unsorted within each. Each range query analyzes the cracker index, scans key ranges that fall entirely within the query range, and uses the two end points of the query range to further partition the appropriate two key ranges.

For example, a read-only query on the range "d – p" would crack the keys "y j s c s g m a q k b u e" into three partitions: (1) "c a b e", (2) "j g m k", and (3) "y s q u." If next a new query with range boundaries j and s is processed, the values in partition (1) could be ignored, but partitions (2) and (3) would be further cracked into partitions (2a) "j", (2b) "g m k", (3a)"q s", and (3b)"y u". Subsequent queries continue to partition these key ranges until the structures have been optimized for the current workload.

Updates and their efficient integration into the data structure are covered in [11]. Multi-column indexes to support selections and tuple reconstructions are covered in [9]. Paper [9] also handles storage restrictions via dynamic partial index materialization.

Adaptive Merging. Inspired by database cracking, adaptive merging [6,7] works with block-access devices such as disks, in addition to main memory. The principal goal for designing adaptive merging is to reduce the number of queries required to converge to a fully-optimized index, and the principal mechanism is to employ variable amounts of memory and CPU effort in each query.

While database cracking functions as an incremental quicksort, with each query resulting in at most two partitioning steps, adaptive merging functions as an incremental external merge sort. Under adaptive merging, the first query to use a given column in a predicate produces sorted runs, ideally stored in a partitioned B-tree [5], and subsequent queries upon that same column perform merge steps. Each merge step affects key ranges that are relevant to actual queries, avoiding any effort on all other key ranges. This merge logic executes as a side effect of query execution.

For example an initial read-only query on the range "d – p" might break the keys "y j s c s g m a q k b u e" into equally-sized partitions and then sort them in memory to produce sorted runs : (1) "c j s y", (2) "a g m q", (3) "b e k u". If next a second query with range boundaries j and s is processed, relevant values would be retrieved (via index lookup) and merged out of the runs and into a "final" partition (fully-optimized for the current workload): (1) "c s y", (2) "a g", (3) "b e u", (final) "j k m q". Subsequent queries continue to merge results from the runs until the the "final" partition has been optimized for the current workload.

Hybrids. Currently, we are actively combining ideas from both database cracking and adaptive merging with the goal of combining the strengths of these adaptive indexing techniques so as to better handle dynamic environments.

2.3 Traditional and Online Index Tuning Benchmarks

Traditional benchmarks consider only the query processing costs. More recently, [14] introduces a new benchmark that captures metrics about online indexing techniques. The main distinction of [14] is that it includes the index creation costs as part of processing performance, and is thus suited for evaluating online techniques that create indexes on-the-fly.

Typically, the cumulative cost is considered, i.e., the total time needed to run a workload of Q queries. For an online indexing technique though, [14] includes the cost to run a number of queries without index support as long as the monitoring and decision phase lasts as well as the costs to create the proper indexes and subsequently the cost to run the rest of the workload with index support. The quality of an online technique is based on minimizing this total time as in the case of a classic indexing technique. In online tuning, however, this can be broken down into the individual phases that characterize the costs and benefits of an online technique. For example, one metric is how fast the system

recognizes which indexes to build. The faster this happens, the more queries can benefit from an improved performance.

To evaluate an adaptive indexing technique properly, we must understand both the benefits and also the overhead costs of the incremental index improvements. In adaptive indexing, indexes are built continuously with every query. Each query runs several operators and each operator performs incremental physical changes that improve the structure of indexes. Thus, indexes are not built at once, but rather in numerous steps that are determined by the workload. In fact, an index may never reach a final, fully refined, state if a less-refined state can provide sufficiently good performance for the current workload. In this way, the setting for evaluating adaptive indexing technique is drastically different. We need to carefully identify and evaluate the multiple different stages that index creation goes through until it reaches a stable status. During each stage, the performance characteristics are different. The duration of each of these stages characterizes the quality of an adaptive indexing technique. In the following section, we discuss these concepts in detail.

3 Framework for Adaptive Indexing Benchmarks

As the authors of two different approaches to adaptive indexing [6,7,9,10,11] and long-time admirers of previous self-tuning approaches [1,2,3,12,13] we propose here a framework for benchmarking adaptive indexing systems. One design goal is that the framework should be able to measure the incremental costs and benefits of reorganization actions in terms of how these are distributed along the lifetime of a workload. A second design goal is that this framework should be generic enough that a variety of utility functions and workloads could be used. For example, an implementation of this benchmark could be used to compare two adaptive indexing techniques on the basis of the time needed to execute the workload just as well as on the basis of the energy used. Finally, the framework should support the comparison of the effectiveness of an adaptive indexing technique to any other indexing technique, whether adaptive, online or traditional. For example, given a utility function and a workload, a database practitioner should be able to determine how an adaptive indexing technique compares to a traditional approach in terms of that workload.

We begin by identifying stages that describe how the costs and benefits of adaptive indexing are distributed along the lifetime of a workload. Distinguishing between these stages informs comparisons between different adaptive indexing techniques. Second, we discuss how general design considerations, such as workload composition and query selection, can impact adaptive indexing performance in each of these stages. Finally, we discuss the use of metrics to compare adaptive indexing performance across the lifetime of a workload.

3.1 Stages of Adaptive Indexing

We define stages of an adaptive indexing life-cycle in terms of the overhead that incremental indexing adds to each query, versus the degree to which that query benefits from the efforts of prior queries.

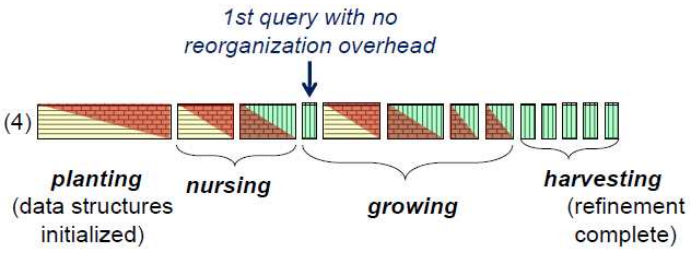

Fig. 2. Adaptive Indexing Stages

As shown in Figure 2, these two metrics help us to identify four stages of adaptive indexing over the lifespan of a workload phase. As starting point, we assume that all data has been loaded into the database system, but no index structures have been created, yet.

Stage 1: Planting. This first stage is characterized by the fact that per-query costs of adaptive indexing exceed those of scan-based query evaluation. The initial step in adaptive indexing is always the extraction of future index entries from the original data collection, e.g., a table stored unsorted and in row format. Even this step can be implemented as side effect of query execution. Subsequently, the index is refined as a side effect of each query. During the planting stage, the expenses for initializing and refining the index exceed the benefits of exploiting the still rudimentary index. Consequently, the total per-query costs are larger than with scan-based query evaluation.

Stage 2: Nursing. With more queries being executed, the index becomes more detailed and complete, yet query execution benefits from the efforts of prior queries. Hence, the expenses for further refining the index decrease while at the same time the benefits of using the improving index increase. Consequently, the per-query costs of adaptive indexing decrease. The point where the per-query costs of adaptive indexing become lower than those of scan-based query evaluation marks the beginning of this second stage. During the nursing stage, the investments of the planting stage start paying-off in terms of per-query costs. However, the cumulative costs over all queries for adaptive indexing still exceed those of scan-based query evaluation.

Stage 3: Growing. As index refinement proceeds, the cumulative benefits of the nursing stage eventually outweigh the cumulative investments during the planting stage. The first query that benefits from the restructuring efforts of previous queries without having to expend any further effort itself beginning of the growing stage, i.e., the stage at which the index structure begins to converge to an optimal state for the current workload phase.

Stage 4: Harvesting. Finally, the index structure is fully optimized and query execution no longer includes side effects. Per-query execution costs reach a minimum. We refer to this final stage as harvesting.

Discussion. The above metrics are drastically different than simply measuring the performance of an a priori fully optimized system or simply considering a one-time index creation online. In adaptive indexing individual queries perform small physical design changes and optimal physical design is reached only after a number of queries have been processed. For adaptive indexing an index is optimal if it allows the current query to be processed in the same time as a fully materialized and fully optimized traditional index. This does not mean though that the adaptive index is completely tuned at this point for the complete data set. It even does not mean that the adaptive index is completely materialized.

Adaptive indexing stages apply to both new non-clustered (secondary, redundant) indexes, as well as to individual key ranges. Applying adaptive indexing to clustered (primary) indexes is more akin to table reorganization rather than index creation. We note that the four stages defined above do not necessarily occur only once per workload, but rather once per index (possibly partial) that a workload phase requires.

While originally defined for adaptive indexing, we can also fit traditional a priori index creation and online index selection/creation into the 4-stages framework. For traditional a priori index creation, the planting stage consists of the actual index creation, and the remainder of the workload moves directly into the harvesting stage. For online index creation, the planting stage covers the initial workload monitoring that leads to the index creation and the index creation itself. After this, the remainder of the workload phase moves directly into the harvesting stage.

3.2 Design Considerations

A benchmark should evaluate the design tradeoffs made by the techniques it evaluates. For example, an online index selection benchmark may test how the allocation of a space budget, the monitoring time period, and the analysis budget impact performance of an index selection technique. In the case of adaptive indexing, because index creation and refinement takes place automatically during the execution of individual queries, there is no monitoring time period, analysis, or even index selection needed. Instead, an adaptive indexing benchmark should test how workload composition and the amount of work performed by query execution side-effects impact each stage of the adaptive indexing process. For a given technique and workload, certain stages might become longer or shorter or even be skipped entirely.

Workload Phases. Because adaptive indexing particularly targets shifting workloads, we model a workload W as a sequence of phases. Each workload phase P comprises a sequence of queries Q and a scheduling discipline S that determines how they will be submitted to the database: $P = (Q, S)$.

Each phase of a workload potentially calls for new index structures and thus passes through the planting, nursing, growing, and harvesting stages. When there is a gradual transition between phases, queries associated with the old phase may be in growing or harvesting stages while queries associated with the new phase

must begin at the planting stage, although it is possible that the preliminary stages of the new phase may be skipped or at least facilitated by work done during a prior phase.

We can model any given indexing mechanism as a transformation function that transforms Phase P's original sequence of queries Q into a new sequence of queries Q' at runtime: $transform(Q, S) = (Q', S)$. Each query $q \in Q$ is transformed individually, depending on its place in the workload.

Utility. Assume there exists a measure of utility $utility(q)$ that applies to the execution of each query and that can also be applied to stages, phases and workloads. For example, one measure might be the time needed for a workload phase to complete: $utility(P) = 1/time(P)$. Other simple measures might be the power used during execution of a query: $utility(q) = 1/power(q)$, or the number of records touched during query execution: $utility(q) = 1/records\_accessed(q)$.

During the planting stage, each transformed query in $q' \in Q'$ has less, or at best equal, utility than its original counterpart. During the nursing stage, some transformed queries have increased utility compared to their original counterparts. At the growing stage, all transformed queries have increased utility dominate those with decreased utility. Finally, during the harvesting stage, the index structure is fully optimized from the perspective of that particular workload phase, and all remaining queries are overhead-free.

Metrics. There are a number of ways to assess the utility of an adaptive indexing mechanism with regard to a given workload. We can assess the overall impact of an adaptive indexing mechanism by comparing the utility of the original and transformed workload. We can compare the overall efficiency of adaptive indexing by comparing the utility of the transformed workload to the utility of a workload with pre-populated indexes.

In addition, because the premise of adaptive indexing is that a workload can reap immediate benefits with low initial investments, we should also consider the cost of the planting and nursing stages, as well as the utility of queries within the nursing and growing stages. To this end, we can measure the aggregate utility per query. Finally, we can consider the speed of convergence (how many queries it takes to reach the harvesting stage).

Experimental Parameters. A number of factors impact how the above metrics are met with regard to a given workload, and that benchmark specifications should consider. The goal of an adaptive indexing benchmark would be to stress an adaptive indexing technique regarding its ability to maintain a fluid and quick transition from one stage to the next. The ideal goal of an adaptive indexing technique is to quickly move through all stages and reach the harvesting stage. The even more crucial part is that it quickly enters the nursing and growing stages so that it can materialize immediate benefits when the workload changes. Thus, critical parameters to study include:

- Varying the range and distribution of keys used in query predicates. Shifting the focus into different key ranges forces an adaptive indexing technique to exit the harvesting stage and return to previous stages. Once the index is again optimized enough or completely for the new key ranges, we again enter the growing and harvesting stage. The smallest the disruption of the stages, the best the adaptive indexing technique is.
- Varying the density of phases per workload rewards strategies that can adapt quickly to a new phase. With workload phases changing rapidly there is less time to spend in adapting so instant reactions and benefits are needed.
- Varying the overlap between workload phases rewards strategies that can leverage effort done during prior phases.
- Varying the number of columns involved in query predicates as well as the tables used in the query workload stresses the ability of an adaptive indexing technique to focus into multiple parts of the data at the same time. It typically extends the length of the stages as it takes more queries and time to improve performance on a given data part. It also stresses the ability of the system to maintain and administer an extended table of contents efficiently.
- Varying the concurrency of queries (the scheduling policy) stresses the ability of an adaptive indexing technique to properly schedule or serialize multiple queries. Ideally, the stages should show the same behavior as if the queries arrive one after the other.
- Varying the percentage of updates in the workload stresses the ability of an adaptive indexing technique to not disturb the stages flow while new data are merged and affect the physical actions performed. At worse an update invalidates all previous actions and leads back to the planting stage. Adaptive indexing techniques though should rely on incremental and differential approaches in order to maintain the stage development.
- Varying the amount of available storage stresses adaptive indexing for its ability to gain and exploit partial knowledge when storage is not enough to accommodate all necessary indexes. It should be able to work on partially materialized indexes and continuously refine them, augment them and reduce them as the workload shifts. Again a powerful adaptive indexing technique is characterized for its ability to quickly go past the planing stage and materialize performance benefits.

4 Partial Reference Implementation

In this section, we illustrate the stages and metrics described in Section 3 using the results of experiments previously published in [9] and [7].

4.1 General Experimental Setup

The ensuing experiments assume no up-front knowledge and, crucially, no idle time to invest in any preparation. Data is loaded up-front in a raw format and queries immediately arrive in a single stream.

The database cracking implementation is built on top of the MonetDB open-source column-store, which resulted in the design of new operators and optimizer rules in the kernel of MonetDB. Experiments were run using a 2.4 GHz Intel Core2 Quad CPU equipped with one 32 KB L1 cache per core, two 4 MB L2 caches, each shared by 2 cores, and 8 GB RAM. The operating system is Fedora 12. The reported experiments for database cracking measure the elapsed execution time for each query processed.

The adaptive merging experiments were done using a simulator capable of configuring experimental parameters such as workspace size, merge fan-in, initial partition size, etc. The metric used is the number of records touched by each query (as opposed to the number of comparisons) which is appropriate for evaluating techniques that target movements in the memory hierarchy.

4.2 Planting, Nursing, and Growing Stages

Our first experiment runs 1000 simple selection queries defined on a 3 attribute table of 10^7 tuples with unique integers randomly located in the table columns. Having a data workload that equally spans across the value domain is again the hardest scenario for adaptive indexing as it offers no flexibility to focus and improve certain areas. The queries are of the following form:

$$\text{select } max(B),max(C) \text{ from R where } v_1 < A < v_2$$

The queries are focused on a particular range of data — we choose v_1 and v_2 such that 9/10 queries request a random range from the first half of the attribute value domain, while only 1/10 queries request a random range from the rest of the domain. All queries request 20% of the tuples.

Figure 3 is based upon Figure 6 of [9], and compares the performance of database cracking (blue), full-sort/traditional indexing (magenta), and scan-only (red) approaches. The utility function is based on elapsed execution time of each query.

The green annotations mark the planting, nursing, and growing stages of database cracking. The nursing stage begins when the first time the cost executing a database cracking query is less than the cost of executing a scan-only query. The growing stage begins with the tenth query — the first that does not incur a cracking action. Note that because the cracking overhead is minimal, in practice, performance at the growing stage eventually matches the full-sort performance. Actually, even in the nursing stage individual query response times are significantly improved over the scan approach and only marginally slower than the presort one.

For the full-sort approach, only the harvesting stage is shown. The presorting cost (planting stage) for the full-sort approach was 3.5 seconds and is not shown on the graph. In other words, this approach assumes perfect knowledge and idle time to prepare. It represents the optimal behavior after paying a hefty cost in the beginning of the query sequence. Since the planting stage fully refines data structures, no nursing or growing stages take place.

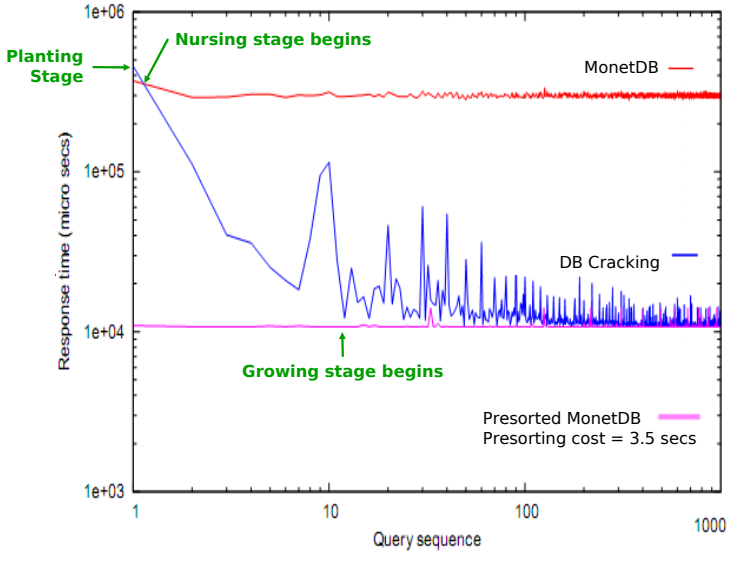

Fig. 3. Adaptive indexing stages illustrated by database cracking

The scan-only approach builds no auxiliary data structures, and thus does not participate in any of the adaptive indexing stages at all. It cannot exploit the fact that the query workload is mainly focused on a small area of the data.

4.3 Shorter Stages

Next we consider the results of an equivalent experiment run using adaptive merging. Adaptive merging is designed with the property of reaching faster the harvesting stage in terms of queries needed. It invests a bit more effort than cracking during the planting stage but less effort than a full sort approach.

This experiment uses a workload of 5,000 queries. Queries are against a random permutation of the integers 0 to 9,999,999. Each query requests a random range of 1 value to 20% of the domain; 10% on average. Initial runs in the partitioned B-tree are created with a workspace of 100,000 records, for 51 initial partitions. The merge fan-in is sufficient to complete all B-tree optimization in a single merge level.

Figure 4 is based upon Figure 9 of [7], and compares the performance of adaptive merging (red), full-sort/traditional indexing (purple), and scan-only (green) approaches. Each data point shows the average of 1% of the workload or 50 queries. Note that in this graph, the utility function is based on the number of records accessed by each query.

The scan and presort options show the same behavior as in the previous experiment. Adaptive indexing though shows a different behavior with the stages being much shorter. The red annotations mark the nursing, and harvesting stages

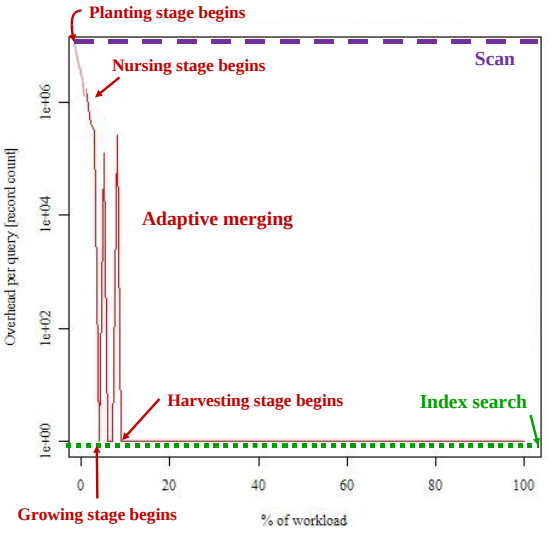

Fig. 4. Adaptive merging query sequence: shorter stages

of adaptive merging. Given the active nature of adaptive merging the harvesting stage begins very fast leaving a fully optimized B-tree after less than 50 queries.

4.4 Multiple Workload Phases, Including Updates

The experiments described above each address only a single phase. Our next two experiments illustrates adaptive indexing in the context of a workload with multiple phases. We first consider a workload consisting of ten phases representing drifting range queries with a drifting focus, as executed by adaptive merging. We next consider a workload consisting of update and read-only query phases, as executed by database cracking.

Drifting Query Focus. One of the design goals of adaptive indexing is to focus index optimization effort on key ranges that occur in actual queries. If the focus of query activity changes, additional key ranges are optimized in the index as appropriate. In this workload, 10^7 records with unique key values are searched by 500 queries in five phases that shift focus from the lowest key range in the index to the highest key range.

Figure 5 is based upon Figure 18 of [8], illustrates the overhead per query as the workload passes through the phases. As in the previous adaptive merging experiment, the utility function is based on the number of records accessed by each query. Because the data accessed by the various phases does not overlap, each new phase must pass through new nursing, growing, and harvesting stages.

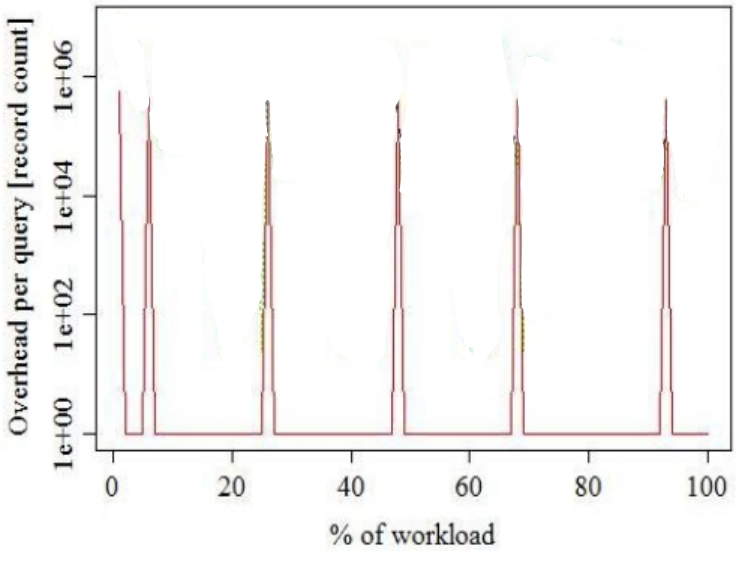

Fig. 5. Workload with five phases of query focus illustrated by adaptive merging

Mixture of Updates and Read-Only Queries. Next we consider a workload that contains a random mixture of update and query phases. Naturally, updates require some auxiliary work which pose an overhead that may eventually disturb the normal flow of adaptive indexing stages.

Two scenarios are considered here, (a) the high frequency low volume scenario (HFLV); every 10 queries we get 10 random updates and (b) the low frequency high volume scenario (LFHV); every 10^3 queries we get 10^3 random updates. Random queries are used in the same form as for the first cracking experiment. Using completely random queries represents the most challenging workload for an adaptive technique as there is basically no pattern to adapt to. In other words, using a random workload will result in the longest possible stages in the adaptation procedure.

Figure 6 is based upon Figure 7 of [9], and shows that cracking maintains high performance and a self-organizing behavior through the whole sequence of queries and updates. Peaks and bumps occur frequently disturbing momentarily the current stage every time.

The power of an adaptive indexing technique is how well it can absorb these peaks. To achieve this an adaptive indexing needs to rely on adaptive and incremental methods for handling updates. In this case, updates are handled during query processing as part of the incremental physical design changes, i.e., the actual query processing operators in the DB kernel are responsible for on-the-fly merging the necessary updates.

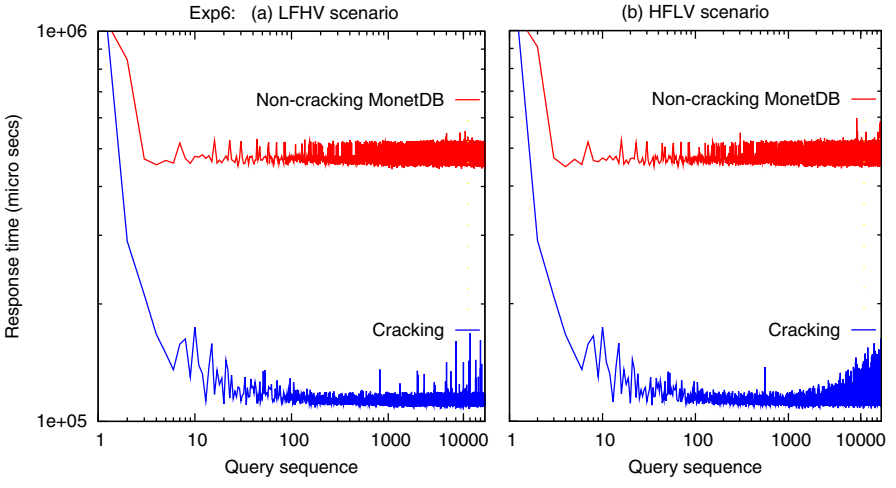

Fig. 6. Effect of updates

5 Outlook

Adaptive indexing techniques (database cracking, adaptive merging, and hybrids) can be combined with automatic index tuning in multiple ways. A tuning tool might prevent certain indexes (e.g., due to excessive anticipated update costs) or it might encourage certain indexes (e.g., for immediate creation as side effect during the first appropriate query execution). Alternatively, the tuning tool might observe activity and pursue index optimization proactively without waiting for queries and their side effects. It might perform only some initial steps of adaptive index creation and optimization (e.g., extraction of future index entries and run generation, but not merging) or it might finish partially optimized indexes (e.g., sort small partitions left by database cracking). In addition, a tuning tool could set resource-usage based policies that limit adaptive indexing during query execution (e.g., based on memory allocation during run generation or merging). We intend to explore in our future research some or all of these combinations of adaptive techniques with traditional index tuning techniques. Benchmarks that measure and compare costs and benefits of such hybrid techniques will increase our understanding and guide database developers when choosing techniques to implement and when guiding the application developers.

6 Summary

In this paper, we have laid out the first framework for benchmarking adaptive indexing techniques. We have described the problem of adaptive indexing, discussed characteristics that differentiate adaptive indexing approaches from alternatives, and proposed a framework for comparing these characteristics. Unlike traditional indexing techniques, adaptive indexing distributes the effort of

indexing incrementally across the workload as a side effect of query execution. Each phase of a workload goes through distinct stages of the adaptive indexing life-cycle in terms of the overhead that incremental indexing adds to each query, versus the degree to which that query benefits from the efforts of prior queries. An adaptive indexing benchmark for dynamic database scenarios must take both workload phases and adaptive indexing stages into account, including stressing the system's ability to maintain a rapid and fluid transition from one stage to the other. For the sake of illustration, we described our partial reference implementation of a benchmark instance using this framework.

Adaptive indexing and the ways to evaluate it represent a completely new paradigm. We believe the new evaluation methods presented here can also be exploited by existing offline and online techniques to improve performance in dynamic scenarios.

References

1. Bruno, N., Chaudhuri, S.: To tune or not to tune? a lightweight physical design alerter. In: VLDB (2006)
2. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In: ICDE (2007)
3. Bruno, N., Chaudhuri, S.: Physical design refinement: the 'merge-reduce' approach. In: ACM TODS (2007)
4. Chaudhuri, S., Narasayya, V.R.: Self-tuning database systems: A decade of progress. In: VLDB (2007)
5. Graefe, G.: Sorting and indexing with partitioned b-trees. In: CIDR (2003)
6. Graefe, G., Kuno, H.: Adaptive indexing for relational keys. In: SMDB (2010)
7. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In: EDBT (2010)
8. Graefe, G., Kuno, H.: Two adaptive indexing techniques: improvements and performance evaluation. In: HPL Technical Report (2010)
9. Idreos, S., Kersten, M., Manegold, S.: Self-organizing tuple reconstruction in column stores. In: SIGMOD (2009)
10. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In: CIDR (2007)
11. Idreos, S., Kersten, M.L., Manegold, S.: Updating a cracked database. In: SIGMOD (2007)
12. Lühring, M., Sattler, K.-U., Schmidt, K., Schallehn, E.: Autonomous management of soft indexes. In: SMDB (2007)
13. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: COLT: continuous on-line tuning. In: SIGMOD (2006)
14. Schnaitter, K., Polyzotis, N.: A benchmark for online index selection. In: ICDE (2009)
15. Tukey, J.W.: Exploratory Data Analysis. Addison-Wesley, Reading (1977)