Self-selecting, self-tuning, incrementally optimized indexes

Goetz Graefe
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

Harumi Kuno
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94304

Abstract

In a relational data warehouse with many tables, the number of possible and promising indexes exceeds human comprehension and requires automatic index tuning. While monitoring and reactive index tuning have been proposed, adaptive indexing focuses on adapting the physical database layout for and by actual queries.

"Database cracking" is one such technique. Only if and when a column is used in query predicates, an index for the column is created; and only if and when a key range is queried, the index is optimized for this key range. The effect is akin to a sort that is adaptive and incremental. This sort is, however, very inefficient, particularly when applied on block-access devices. In contrast, traditional index creation sorts data with an efficient merge sort optimized for block-access devices, but it is neither adaptive nor incremental.

We propose *adaptive merging*, an adaptive, incremental, and efficient technique for index creation. Index optimization focuses on key ranges used in actual queries. The resulting index adapts more quickly to new data and to new query patterns than database cracking. Sort efficiency is comparable to that of traditional B-tree creation. Nonetheless, the new technique promises better query performance than database cracking, both in memory and on block-access storage.

Categories and subject descriptors

E.2 Data storage representations – arrays, sorted trees.

Keywords

Database index, adaptive, autonomic, query execution. [1]

1 Introduction

In a relational data warehouse with a hundred tables and a thousand columns, billions of indexes are possible, in particular if partial indexes, indexes on computed columns, and materialized views with their indexes are considered. Thus, index selection is a central, classic, and very hard problem in physical database design. Too few or the wrong indexes force many queries to scan large parts of the database; too many indexes force high update costs. Unpredictable ad-hoc queries exacerbate the problem.

One approach is to focus on enabling very fast scans, e.g., using shared scans and columnar storage formats, an approach suitable to high-bandwidth high-latency devices such as traditional disk drives and disk arrays. Low-latency database storage such as flash memory will likely re-energize research into index-based query processing.

Another approach is to tune indexes in response to the actual workload. Contemporary index selection tools rely on monitoring database requests and their execution plans, occasionally invoking creation or removal of indexes on tables and views. Such tools tend to suffer from three weaknesses. First, the interval between monitoring and index creation can exceed the duration of a specific request pattern; in which case there is no benefit to those tools. Second, even if that is not the case, there is no index support during this interval, so data access during the interval is wasted with respect to index creation, and eventual index creation imposes an additional load that interferes with query execution. Last, but not least, traditional indexes on tables cover all rows equally, even if some rows are needed often and some never. For example, recent business transactions are queried more often than those years ago, extreme price fluctuations are more interesting than stable prices, etc. Even where it is possible to limit an index, e.g., using a partial index or a materialized view, it is often difficult or impossible to predict the key ranges to focus on.

Database cracking [IKM 07a, KM 05] has pioneered focused, incremental, automatic optimization of the representation of a data collection – the more often a key range is queried, the more its representation is optimized. This optimization occurs entirely automatically, as a side effect of queries over key ranges not yet fully optimized.

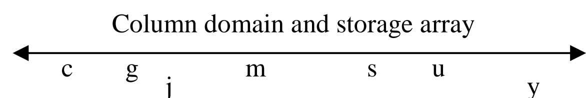

Figure 1. A column store partitioned by database cracking.

For example, after the column store illustrated in Figure 1 has been queried with range boundary values c, g, m, s, and u, all key values below c have been assigned to

---

[1] Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2010, March 22-26, 2010, Lausanne, Switzerland.
Copyright 2010 ACM 978-1-60558-945-9/10/0003 ...$10.00.

storage locations to the far left, followed by all key values between c and g, etc. When a new query with range boundaries j and y is processed, the values below g are ignored, the values between g and m are partitioned with pivot value j, the values between m and u are returned as query results without partitioning or sorting, and the values above u are partitioned with pivot value y. Subsequent queries continue to partition these key ranges.

Database cracking was a distinct innovation when it was proposed, but it also has some weaknesses. First, database cracking requires many steps to reach the final representation for a key range, even if no further cracking is applied to minimal partitions smaller than a pre-determined size, say 4 MB. Thus, optimization of newly loaded data and adaptation to a new focus of query activity are much slower and more expensive than they ought to be. Second, the efficiency of transforming an initial data representation into a fully optimized one depends on the query pattern, i.e., the sequence in which boundary keys and thus pivot values are introduced. Third, search efficiency never reaches that of a traditional index if cracking leaves unsorted minimal partitions, say 4 MB. While a traditional index permits binary search with log2(N) comparisons among N records, the expected cost for a linear search in an unsorted minimal partition is N/2 comparisons (assuming foreknowledge of precisely one match). For example, with 65,000 records in a partition (4 MB ÷ 60 B), the difference is 16 versus 32,500 comparisons or a factor 2,000. Fourth, database cracking seems to work well for in-memory databases but not for block-access storage.

In this paper, we introduce adaptive merging, a new technique that overcomes these weaknesses. It combines the efficiency of traditional B-tree creation with the adaptive and incremental behavior of database cracking. It differs from database cracking as it is based on merging (as in a merge sort) rather than on partitioning (as used in quicksort). The performance advantages of adaptive merging are substantial both during adaptation when the focus of query activity shifts to a new key range and during individual queries against a fully optimized data representation.

2 Prior work

Our design relies on multiple directions of prior work. Section 2.1 covers approaches that monitor execution and periodically evaluate what indexes are appropriate for a running workload. Section 2.2 reviews database cracking, the only other approach we know of that continually adapts index structures to reflect a running workload. Finally, Section 2.3 describes one of multiple mechanisms well-suited to providing an underlying storage mechanism for implementing our adaptive merging approach.

2.1 Automatic index selection

As indexes are crucial to query performance in most database systems, the problem of selecting the best index set is as old as physical data independence. Most previous approaches have focused on automating decisions of which indexes to create, merge, or drop [BC 07, CN 07, FST 88, H 76, S 74]. Both index tuning and index creation costs are added to the database workload. Once a decision is made, it affects all key ranges in an index. For example, Bruno and Chaudhuri maintain a set of statistics about incoming queries, use them to identify and evaluate existing and candidate indexes, and then explicitly create or drop target indexes [BC 07]. They do not consider partial indexes, nor do they support partially completing an index to focus efforts on the key ranges of interest in the query workload, and to avoid effort spent on other key ranges.

The recognition that some data items are more heavily queried than others has led to the concept of partial indexes [S 89, SS 95]. A generalization is the concept of materialized views and their indexes. This results in further increases in the complexity and run-time of index tuning.

The present research is orthogonal to the contents of indexes as it focuses on mechanisms for dynamic creation, optimization, and maintenance of a general search structure. We propose a mechanism that, like database cracking, can automatically create and refine index structures as required by the current pattern of selection queries. Note that our approach is complementary to "monitor queries then build indexes" approaches, in that our method could be invoked when observations indicate a potential benefit. That is to say, adaptive indexing could provide mechanisms and could be guided by the observe-and-tune techniques proposed in those earlier research efforts.

2.2 Database cracking

Database cracking, which combines features of both automatic index selection and partial indexes, is the only other proposal we have seen for refining index structures as a side effect of unpredictable dynamically-arriving queries. When a column is used in a predicate for the first time, a cracker index is created by copying all data values in the appropriate column from the table's primary data structure. When the column is used in the predicate of another query, the cracker index is refined as long as a finer granularity of key ranges is advantageous.

The keys in a cracker index are partitioned into disjoint key ranges and unsorted within each. As illustrated in Figure 2, each range query analyzes the cracker index, scans the 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. Thus, in most cases, each partitioning step creates two new subpartitions using logic very similar to the partitioning step in quicksort [H 61]. A range is partitioned into 3 new subpartitions if both end points fall into the same key range. This happens in the first partitioning step in a cracker index (because there is only one key range encompassing all key values) but unlikely thereafter [IKM 07a].

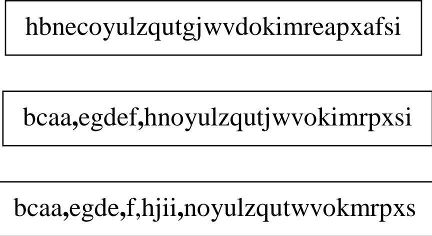

Figure 2. Partitioning in a cracker index.

Figure 2 illustrates how database cracking refines the array of key values in a cracker index. Each character represents a record; the character is its key value. In this small example, each vowel appears twice and each consonant once. The top box shows the initial cracker index immediately after copying. The center box shows the cracker index after a range query for keys d through g. Both end points are included in the new center partition. The bottom box shows the partitions after a subsequent range query for key range f through j. The two new boundary keys are used to further refine two partitions. Partition sizes are very unevenly distributed.

Key ranges never queried are never partitioned or optimized. This is a crucial advantage of adaptive indexing over traditional indexes including those created by most external tuning tools. On the other hand, each individual data record is moved many times during the incremental transformation from the un-optimized initial representation to the fully optimized final representation.

As the core operation of database cracking is very similar to partitioning in quicksort [H 61]. In an in-memory database, it performs very well. As each key value in the database must be compared with the boundary key of the query predicate, it adds very little to the elapsed time of each query. In a sense, database cracking is designed to achieve the maximal benefit in the data organization without slowing down in-memory scans, i.e., to maximize the benefit that can be achieved "for free."

With block-access devices like disk and flash storage, it probably performs like quicksort in virtual memory, which database systems do not use. For block-access storage, database developers have long favored external merge sort and B-tree indexes. As shown in Section 4, adaptive merging performs well over block access storage, requiring an order of magnitude fewer queries than database cracking to optimize an optimized index for a query set.

2.3 Partitioned B-trees

Our search for an adaptive indexing technique suitable for block-access storage led us to an algorithm based on merging rather than on partitioning and to a data structure known as partitioned B-trees [G 03], which are variants of traditional B-tree indexes [BM 72]. Our proposal applies to hash indexes if those are B-trees on hash values and to multi-dimensional indexes if those use a space-filling curve such as UB-trees [B 97]; whether it applies to other hash indexes and other multi-dimensional indexes is left to future research. It applies to indexes on block-access devices such as traditional disks and flash storage as well as to in-memory indexes optimized for CPU caches. Finally, it applies to both primary and secondary B-tree indexes, including multi-column ("compound") B-tree indexes, i.e., to the vast majority of indexes used in practice today.

Partitioned B-trees differ from traditional B-trees as they add an artificial leading key field. Distinct values in this field define partitions within the B-tree. Partitions appear and disappear due to record insertion and deletion, with no catalog modification. Records with the same value in this field can be searched as efficiently as in a traditional B-tree. The desired steady state is to have only a single partition. Temporary additional partitions enable optimizations during index creation, roll-in (loading), and roll-out (purging). Moreover, external merge sort can store runs in B-tree partitions with benefits for deep read-ahead, pause-and-resume, dynamic resource management, etc. Reorganization and optimization from multiple partitions to a single one uses the same merge logic as traditional merge sort.

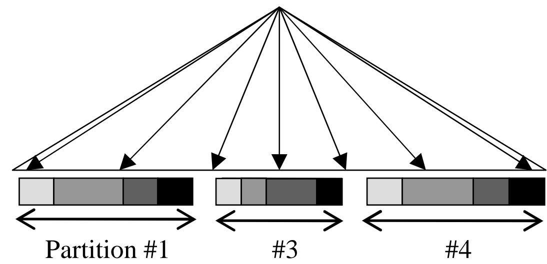

Figure 3. Partitioned B-tree and search pattern.

Figure 3 illustrates a partitioned B-tree (the root at the top and leaves along the bottom) with partitions (each a sequence of shaded boxes indicating key ranges) identified by an artificial leading key field in each record (shown as Partition #). A query enumerates the partitions and searches each one. In the most naïve implementation, a root-to-leaf probe is required to find the next actual value in the artificial leading key field and another probe is required to apply the query predicate within the partition [LJB 95]. The number of probes can be reduced to one per partition.

The fact that adaptive merging exploits partitioned B-trees is incidental; our starting point was a quest to find a technique that is self-tuning similar to database cracking but with better query execution performance both during index optimization and when querying the final data structure. The introduction of partitioned B-trees [G 03] suggested some related techniques but did not suggest optimizing key ranges as a side effect of query execution, the core of both database cracking and adaptive merging.

3 Adaptive merging

Adaptive merging, the technique introduced here, aims to combine efficient merge sort with adaptive and incre-

mental index optimization. Like database cracking, it requires a flexible underlying storage structure for partially and locally optimized index states. Partitioned B-trees appear to be an ideal choice.

The essence of partitioned B-trees, as described above, is to use standard B-trees to persist intermediate states during an external merge sort, to provide efficient search at all times even before B-tree optimization is complete, and thus to separate run generation and merging into independent activities with arbitrary intermediate delay. Partitioned B-trees can also capture intermediate states during index creation, data loading, view materialization, etc.

The essence of adaptive merging is to exploit partitioned B-trees in a novel way, namely to focus merge steps on those key ranges that are relevant to actual queries, to leave records in all other key ranges in their initial places, and to integrate the merge logic as side effect into query execution. Thus, adaptive merging is like database cracking as it is similarly adaptive and incremental but they differ fundamentally as one relies on merging whereas the other relies on partitioning, resulting in substantial differences in the speed of adaption to new query patterns.

The differences in query performance are due to data being kept sorted at all times in a B-tree. The difference in reorganization performance, i.e., the number of queries required before a key range is fully optimized, is primarily due to merging with a high fan-in as opposed to partitioning with a low fan-out of 2 or 3. The following sections explain in more detail.

3.1 Index selection

For index selection, our design copies the heuristic from database cracking: When a column is used in a predicate for the first time, a new index is created by copying appropriate values. Refinements such as external guidance which indexes to avoid and which ones to choose with priority in queries with multiple predicates, partial indexes, multi-column indexes, consideration of other predicates and their desirable indexes, etc. apply quite similarly to both techniques. The record formats are also similar unless compression is used, e.g., for duplicate key values.

The ordering of data records in an initial copy, however, are quite different due to partitioning in database cracking versus merging in our approach.

3.2 Initial index creation

The initial format of a partitioned B-tree consists of many partitions. Each partition is sorted, but the partitions most likely overlap in their key ranges. Subsequent merging brings the B-tree closer to a single sort sequence in a single partition, as described later.

The initial creation of a new partitioned B-tree performs run generation using an in-memory algorithm such as quicksort or replacement selection. The advantage of the latter is the opportunity for runs larger than the memory allocation during initial index creation. Each run forms a partition in the new B-tree.

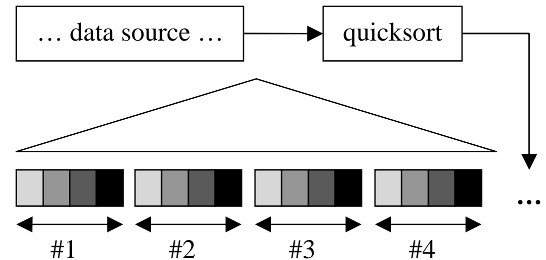

Figure 4. Appending partitions during initial index creation.

Figure 4 illustrates the data movement during initial index creation. Boxes with the same shading indicate the same key range. A run generation algorithm such as quicksort is used to append as many partitions as necessary. Their number depends primarily on input size and memory allocation but also on sort algorithm and any incidental correlation between the sort order in the data source and in the new index.

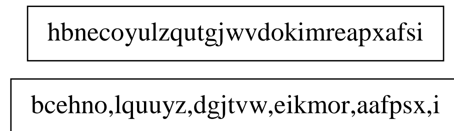

Figure 5. Unsorted input and initial sorted partitions.

Figure 5 shows a concrete example, with the same data as shown in Figure 2. The upper box shows the input, entirely unsorted. The lower box shows the initial index, i.e., records and partitions within a partitioned B-tree. A "," (comma) separates sorted partitions. Run generation during copying produces runs of 6 records in this small example.

Search performance immediately after index creation depends on the count (and thus the average size) of the partitions in the partitioned B-tree, as does the break-even point between probing each partition with a traditional B-tree search and an end-to-end scan of the index. For example, if scan bandwidth is 100 MB/s and each probe takes 20 ms, partitions larger than 100 MB/s × 20 ms = 2 MB ought to be probed rather than scanned, corresponding to a modest memory allocation of 1 MB during run generation by replacement selection. Note that the "lock footprint" can be smaller during probing than during scanning, further favoring probing over scans. Modern flash storage also favors probing over scans. Nonetheless, scanning is always possible if desired, e.g., in order to exploit shared scans.

3.3 Incremental index optimization

When a column is used in a predicate for the second time, an appropriate index exists, albeit not yet fully optimized and merged into a single partition. In this situation, a query must find its required records within each partition, typically by probing within B-tree for the low end of the query range and then scanning to the high end.

Instead of just scanning the desired key range one partition at a time, however, the query might as well scan multiple partitions in an interleaved way, merge these multiple sorted streams into a single sorted stream, write those records into a new partition within the partitioned B-tree, and also return those records as the query result. The data volume touched and moved is that of the query result.

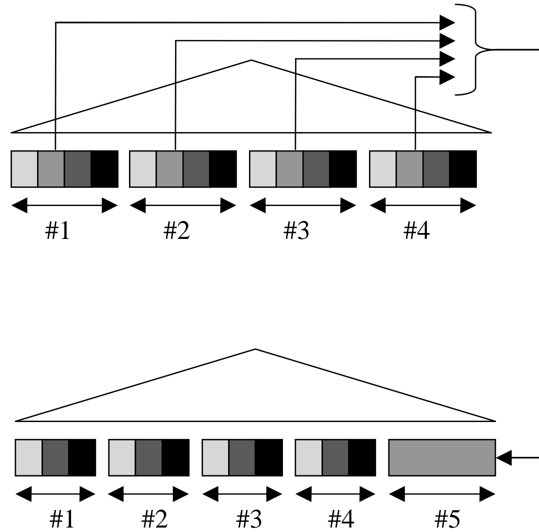

Figure 6. Partitioned B-tree before and after a query.

Figure 6 illustrates merging and data movement during the second query. The top half shows the B-tree when the query starts. In processing the query, records satisfying the query predicate are automatically merged into the new partition, as shown in the bottom half of Figure 6. Ideally, a single merge step suffices to merge records from all existing partition into a single, final partition.

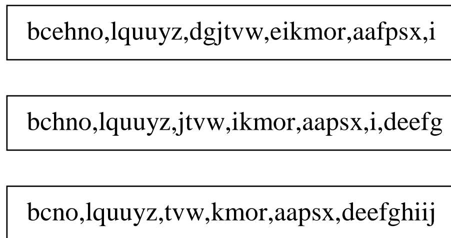

Figure 7. Merging as side effect of query execution.

Figure 7 continues the concrete example of Figure 5, using the same data and range queries as Figure 2. The top box shows sorted partitions after run generation. The center box shows the partitions after a range query for keys d through g. A subsequent query for key range e through f would access only the new partition. The bottom box shows the partitions after subsequent range query for keys f through j. Some of this range can be answered from the merged partition and some of it causes additional merge activity. Note that the smallest among the original sorted partitions vanishes as side effect of the second query. Also note that the merge activity during the second and third queries inserts data into the same partition. This is easily possible as their merge steps focus on disjoint key ranges.

If all existing partitions can be merged to form a single partition in a single step, i.e., the number of initial partitions is smaller than the merge fan-in as limited by the memory allocation available for merging, then the query may leave the keys within its query range in a single location comparable to a traditional, fully optimized B-tree index.

If the query range of the third query is a subset of that of the second query, the third query can search as efficiently in a partitioned B-tree as in a traditional B-tree. If the query ranges of the second and third queries do not overlap, it leaves its result behind in the same format as the second query for the benefit of the future queries. In this case, multiple queries can merge their output into the same new partitions.

Actually, this logic applies to individual key ranges. If the range predicates of the second and third queries overlap partially, the third query needs to split its key range into overlapping and non-overlapping sub-ranges. For overlapping key ranges, the third query finds all data in a single location comparable to a traditional B-tree. For non-overlapping key ranges, it probes all existing partitions, extracts the required records, merges them, and moves them to a new partition, as shown in Figure 6.

All subsequent queries also must analyze their range predicates for overlap with prior queries and the merge effort they applied to the index. Once all records within a key range have been merged into a single partition, subsequent queries in that key range work and perform like queries using a traditional B-tree index.

Key ranges without query activity are never reorganized or merged. Those keys remain in the initial runs produced by run generation. Thus, as in database cracking, no effort is wasted on inactive key ranges after the initial copy step. By adaptively merging only those key ranges actually queried, and by performing merge steps as side effects of query execution, adaptive merging preserves the main strength of database cracking. The main difference is in the speed of adaptation, i.e., the number of times each record is moved before it is in its final location.

If more than a single merge step is required to transform the B-tree index from many initial partitions into a single final partition, each key range must be searched and merged by multiple queries before it is in its final, completely optimized format.

The number of merge steps for each key range is equivalent to the merge depth in an external merge sort, i.e., $\log_F(W)$ for W initial runs merged with fan-in F. With the memory sizes of modern computers, sort operations with a single merge step are common, and sort operations with more than two merge levels are quite unusual. Just as in external merge sort with optimized merge patterns, the

merge depth may not be uniform for all records and thus the average merge depth might be a fraction, e.g., 1¾.

In fact, the number of merge steps per record is a key difference between database cracking and adaptive merging. The merge fan-in can easily exceed 100, whereas the partitioning fan-out in database cracking is usually 2 or 3, limited by the number of new partitioning keys available in any one range query. Thus, database cracking may move each record many times before its final place is found. The exact number depends on the size of partitions to which no further cracking is applied and the size of the initial partitions in the proposed design.

For example, if the size of the cracked index is 1 GB, if partitions smaller than 4 MB are no further partitioned, and the partitioning fan-out is 2, no further partitioning is required for a key range after $\log_2$ (1GB ÷ 4MB) = $\log_2$ 256 = 8 partitioning steps affecting this key range – even more if skew is an issue. In partitioned B-trees with adaptive merging, if the average size of the initial runs is 16 MB and the merge fan-in is 64, then the number of merge levels is $\log_{64}$ (1 GB ÷ 16 MB) = $\log_{64}$ 64 = 1. In other words, in this example, database cracking moves each record 8 times before its final location is found, whereas adaptive merging requires just a single move. Other numeric examples produce similar differences as long as the merge fan-in is much larger than 2.

3.4 Table of contents

As in database cracking, an auxiliary data structure is required to retain information about reorganization efforts already completed. In fact, the set of keys is the same in the auxiliary data structures for database cracking and for adaptive merging. The information associated with those keys differs. In database cracking, the start position of the partition with the given key is indicated. In adaptive merging, the data structure indicates the range of identifiers for partitions with records in the given key range.

For example, suppose that run generation creates runs with identifiers 1 through 1,000. All key ranges can be found in this set of partitions. After a key range has been merged once, say with merge fan-in of 32, records within this key range can be found in partitions 1,001 through 1,032 but not longer in partitions 1 through 1,000. A key range merged twice can be found only in partition 1,033. Query performance in such key ranges equals traditional B-trees.

3.5 Transaction support

As the proposed structure is a B-tree, even if an artificial leading key field is added, all traditional methods for concurrency control, logging, and recovery apply.

In addition, key prefixes could be locked, a generalization of Tandem's "generic locks" [G 07]. When a conflict arises, a merge step can be committed immediately because merge operations do not change the contents of the index, only its representation.

The logging volume during merge operations can be reduced to allocation-only logging. In this mode of operation, the page contents are not logged during merge steps, neither deletions in the merge inputs nor insertions in the merge output. Deletion of individual records can be implemented as updates from valid records to "ghost" records (also known as pseudo-deleted records). A single small log record suffices for multiple records. Deletion of entire pages can be captured by a single small log record. Insertion of new pages requires that the new pages be flushed to disk before the data sources for the page contents are erased, i.e., before committing a merge step.

3.6 Updates

For insertions, deletions, and record modifications, there are each multiple alternative techniques. In each case, the first technique is similar to traditional techniques whereas the second one is optimized for efficient completion of many small transactions.

Insertions can be placed either into the final target partition or they can be gathered in a new partition dedicated to gathering insertions. This partition ought to remain in the buffer pool such that all insertions only update in-memory data structures (other than the recovery log). Multiple new partitions may be added over time.

Deletions can either search for the appropriate record in the index, in whatever partition it might be found, or they insert "anti-matter" quite similar to the "negative" records employed during maintenance of materialized views and during online index creation.

Modifications of existing records can be processed either as traditional updates after an appropriate search or they can be processed as pairs of deletion and insertion, with alternative processing techniques as outlined above.

If insertions, deletions, or updates create new partitions in a B-tree, i.e., introduce new partition identifiers, those partitions and their key ranges must be reflected in the table of contents such that subsequent queries search in all appropriate partitions.

3.7 Variations

Several variations and optimizations are possible beyond the design described so far. This section lists some ideas; we have not yet analyzed them for their true practical value or their precise performance effects.

First, the basic idea seems well suited to capturing and indexing continuous streams, in particular if multiple independent indexes are desired for a single stream. Incoming records are always appended to all indexes in partitions formed by run generation. Continuous "trickle updates" in data warehouses are a special case of streams that can be indexed using the proposed techniques.

Second, the general technique applies not only to disk-based databases but also to databases on flash devices and even to in-memory databases. The resulting differences are quantitative rather than qualitative. For example, due to very fast access latency, smaller page sizes are optimal for flash devices, resulting in higher merge fan-in with a fixed memory allocation and thus fewer merge levels from initial runs to a final, fully optimized B-tree [G 07b]. For in-memory databases, optimization of cache faults leads to run generation within the cache and explicit merge steps to form memory-sized indexes [NBC 95].

Third, partitioned B-trees are useful not only for efficient search but also for efficient query execution with merge joins, "order by" clauses, etc. The final merge activity in the query is precisely equivalent to B-tree optimization, and the merge output can replace the previous partitions with a single, fully optimized partition. For orderings on B-tree fields other than the leading field, a general mechanism comparable to MDAM [LJB 95] seems possible but has not yet been described in the literature.

Fourth, adaptive merging in combination with partitioned B-trees provides mechanisms for dynamically adjusting query costs for the purpose of workload management. During index creation, it is possible at any time to defer the remaining key range within the data source. Doing so speeds up the current query but leaves the new index only partially populated. During index optimization, it is possible at any time to reduce the fan-in of merge steps or to interrupt all merge activity in order to defer some merge effort to later queries. Doing so frees up memory (merge input buffers) and speeds up the current query but fails to optimize the key range for subsequent queries.

Fifth, B-tree optimization and partition merging does not depend on queries. Instead, any idle capacity can be used to optimize a partitioned B-tree for future queries. Adaptive merging can focus on those key ranges that have been queried once but are not yet fully optimized. Database cracking, in contrast, cannot exploit prior queries during idle times because it requires a new partitioning key for each additional step.

Finally, instead of merging the precise key range of a query, the logic could be modified to consume entire B-tree leaves. Space management would become simpler and more efficient, whereas the table of contents would become more complex. Consequently, determining the required partitions during query execution would also be more complex. As a compromise, one can extend a query range to the next "short enough" separator key, quite similar to the key optimizations in suffix truncation (compression) [BU 77]. For example, if the query range starts with "Smith," the merge could start with "Sm". Even an equality query could merge an appropriate key range, for example all keys starting with "Sm". If suffix truncation is applied during B-tree construction, the probability is high that merge range coincides with boundaries between leaf pages in all input partitions. In fact, such a policy might be very useful to avoid an excessive number of small merge steps and thus to ensure efficient adaptation of an index to a new query pattern. If multiple merge levels are required, the heuristics might differ among the levels in order to avoid repeatedly searching a large number of initial partitions. The experiment below extends each merge range in both directions to a multiple of the largest power of two smaller than the width of the query range.

4 Performance evaluation

Here, we focus on a comparison of database cracking and adaptive merging. These two adaptive indexing techniques can be compared experimentally with concise, focused experiments. A comparison with index tuning techniques that run workload analysis and index creation externally and in addition to query execution requires a complete system, a representative physical database design as starting point, and a representative workload, all of which are not available to us at this time.

Our first experiment simulates 50 queries 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. Cracking stops with partitions of 1,000 values. 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.

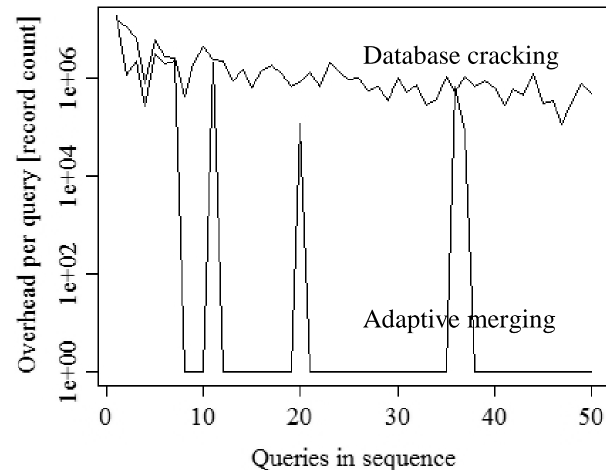

Figure 8. Database cracking and adaptive merging.

Figure 8 shows the overhead effort of database cracking and of adaptive merging. Each query must scan records to produce its output; those are not reflected in the diagram and must be added to compute the total effort. The cost scale in Figure 8 is logarithmic. Note that our cost metric is focused on movements in the memory hierarchy and on the number of records touched; it does not reflect the number of comparisons. Both techniques essentially implement sort algorithms O (N log N) comparisons.

The upper curve shows database cracking. It reflects the behavior known from an earlier performance study [IKM 07a]. The first query partitions the entire domain and thus its overhead is high. The overhead then decreases slowly. Each range query provides two more boundary keys between partitions; thus, the number of partitions in a cracker index is about twice the number of queries processed so far. Dividing 10,000,000 records into partitions no larger than 1,000 records requires at least 9,999 partitioning keys. With at most two new partitioning keys per query, partitioning requires at least 5,000 queries.

The lower curve shows adaptive merging. It converges much faster than database cracking. Merging is practically complete after about 40 queries, meaning the B-tree is fully optimized. In all subsequent queries, search performance equals that of a traditional B-tree. When each query scans precisely the key range needed as query output, the overhead is zero (shown as 1 due to the logarithmic scale).

With a smaller memory allocation during run generation or during merging, multiple merge levels would have been required. For example, convergence would take twice as long with a merge fan-in of only 8 and thus 2 merge levels. Nonetheless, even 100 queries for full convergence would be much faster than database cracking with its strict binary partitioning and thousands of queries prior to steady state. Overhead and convergence rate also depend on the sizes of query ranges as well as skew.

An alternative perspective on the experiment of Figure 8 focuses on the "sortedness" of intermediate states of the index, which may be measured in the number of "adjacent inversions," i.e., neighboring keys with the wrong order relationship. Database cracking enforces no sort order within each partition; thus, each query reduces the number of adjacent inversions only at the new boundaries between partitions, at most two per range query. Adaptive merging permits adjacent inversions only between partitions, and thus even the initial state immediately after run generation has relatively few adjacent inversions.

A third perspective focuses on the relative rather than the absolute overhead in query execution. Key ranges for queries were chosen between 1 value and 20% of the domain, with an average of 10% of the domain or 1,000,000 distinct values, equal to 1,000,000 records in this database. Thus, an overhead of 100,000 records is equal to an extra scan effort of 10% on average, which might seem acceptable in an adaptive indexing technique. From this point of view, database cracking might reach acceptable performance after about a dozen queries. Adaptive merging, on the other hand, always scans only as much data as is required for the query at hand. Merging, if required, can achieve very high bandwidth if the units of disk transfer are sufficiently large, e.g., 1 MB or more.

Figure 9 shows the same experiment as Figure 8 but run over a workload of 5,000 queries. Each data point shows the average of 1% of the workload or 50 queries. Database cracking slowly reduces the overhead per query, although it takes many queries before incremental index optimization ceases. Adaptive merging leaves a fully optimized B-tree after less than 50 queries.

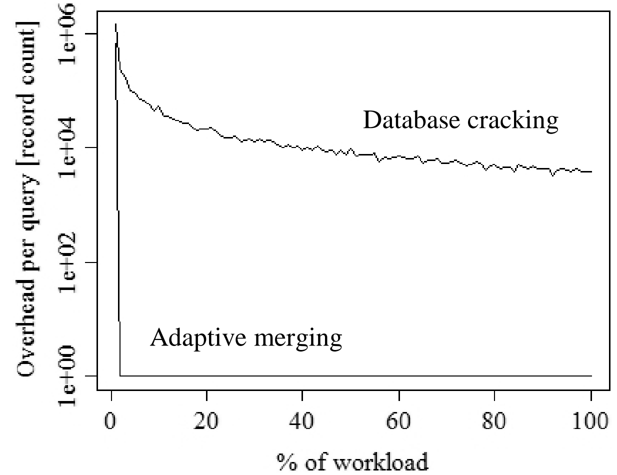

Figure 9. Long query sequence.

If the average query range is small, however, this argument might no longer pertain. For example, if the average query requests only 0.1% (rather than 10%) of the domain, it takes hundreds of queries to achieve acceptable overhead on average, with no guarantee for the worst case. Note that the convergence rate of database cracking should barely affected by the width of the ranges in the queries – each query introduces two new partition boundaries.

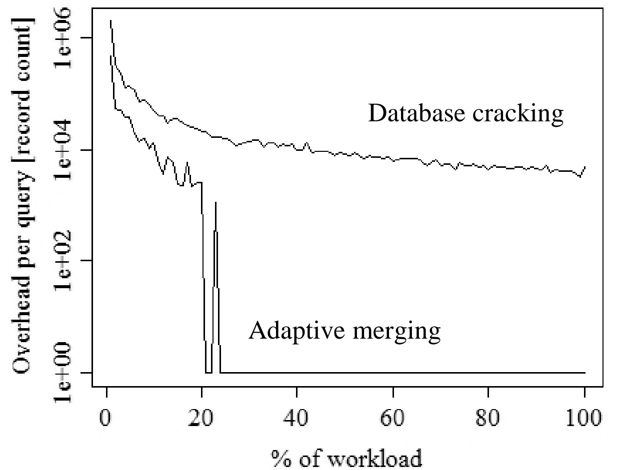

Figure 10. Small query ranges.

Figure 10 shows the average per-query overhead for queries with small ranges. Query predicates range from 1 value to 0.2% of the domain, or 0.1% on average. In other words, this is the same experiment as the prior one except that the average query result size equals 10,000 records rather than 1,000,000 records. Database cracking and its convergence behavior are not affected by the query size; two new partition boundaries with each range query require about 5,000 queries to divide an index with 10,000,000 unique values into minimal partitions of 1,000 records.

Adaptive merging requires more queries than in the prior experiment, but it still optimizes the B-tree over about 1,250 queries (25% of 5,000 queries).

In fact, for the speed of index optimization to be similar in database cracking and adaptive merging, the queries would have to be point queries rather than range queries. If key ranges to be merged are rounded as proposed above as an obvious improvement to the basic technique, it always finishes the index faster than database cracking.

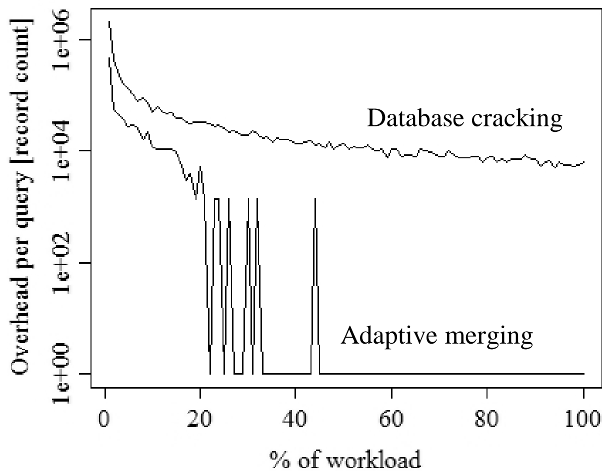

Figure 11. Very small query ranges.

Figure 11 shows the same experiment again with average query result size of 100 rather than 10,000 records. The convergence behavior of database cracking is again unchanged, as expected. The convergence rate of adaptive merging would suffer without an implementation heuristic we found necessary. Inspired by B-trees on block-access devices, our implementation merges a larger key range than is required by the query at hand. Rounding depends on the size of the workspace and the prescribe merge fan-in, based on the assumption that the merge fan-in as the quotient of workspace size and block size. With this optimization, adaptive merging finishes optimization of the B-tree with less than 2,500 queries (50% of 5,000 queries).

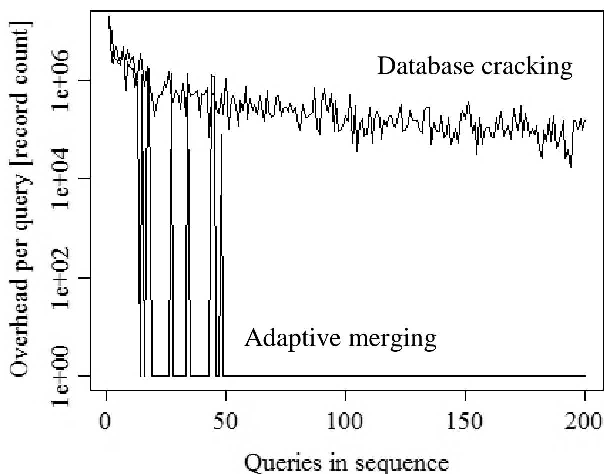

Figure 12. Small memory allocation.

Figure 12 illustrates the effect of a small memory allocation. This experiment equal that of Figure 8 except for the workspace size, the merge fan-in, and the number of queries in the workload. Compared to the prior experiment, both the workspace during run generation and the fan-in during merge steps are reduced by a factor of 10. The workload consists of 200 queries to illustrate the effects. Database cracking is very little affected – it is not designed to exploit a large memory even when memory is readily available. Adaptive merging requires more effort than in the experiment of Figure 8. Each record must go through 3 merge steps before reaching its proper place in the final partition. Nonetheless, adaptive merging converges towards the final index format with fewer moves per record than database cracking. In other words, among the two adaptive indexing schemes, adaptive merging requires much less reorganization effort, whether or not memory is plentiful.

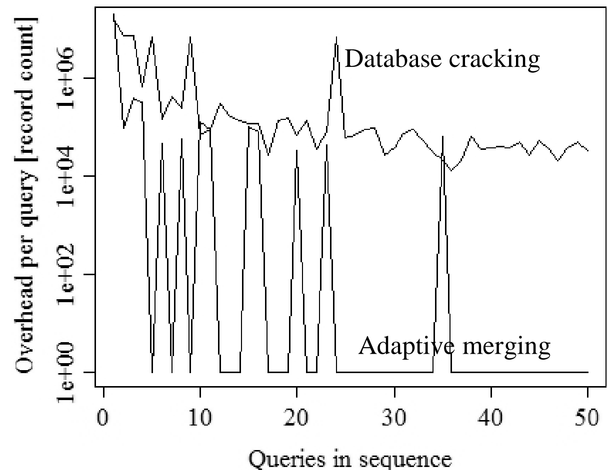

Figure 13. Small query focus.

Finally, Figure 13 illustrates the effect of all queries focusing on the same fraction of the domain, i.e., the case in which adaptive indexing methods are supposed to shine. Here, all queries focus on the 1,000,000 key values in the center of the domain. Otherwise, this experiment equals that of Figure 8. Both adaptive indexing techniques perform better than in the base case shown in Figure 8. However, database cracking imposes more overhead than adaptive merging because it takes many queries before the cracker index attains its final form whereas adaptive merging converges after only 35 queries due to sorted runs and merging with high fan-in. Moreover, the overhead of database cracking exhibits occasional spikes when a query key forces partitioning of the keys above or below the query focus. For example, if the highest key of interest in all queries so far is 5,490,000, a new query with range boundary of 5,491,000 forces partitioning of all records and key up to the 10,000,000.

5 Summary and conclusions

In summary, database cracking and adaptive merging offer a promising alternative to traditional index tuning that relies on monitoring, offline what-if analyses, and long-running index creation utilities that can disrupt processing of the current workload.

While database cracking was designed for in-memory arrays, adaptive merging enables automatic creation and incremental improvements of indexes in large data warehouses on "external" block-access storage. It uses a standard data structure, B-trees, with only a few non-traditional improvements. First, an artificial leading key field permits creation and removal of partitions by insertion and deletion of records with specific partition identifiers; well-known B-tree access algorithms permit efficient search in such partitioned B-trees. Second, index creation is divided into run generation and merging. Both can be side effects of query execution or other scans over the data. Run generation uses a single pass over the future index records and results in a complete, coherent, and searchable B-tree index, even if it is not yet fully optimized. Third, query execution may optimize such an index by merging the key ranges required to answer actual queries, with no effort spent on any unused key ranges. Fourth, those non-optimized key ranges are automatically in a format that can readily be searched and optimized later if the query pattern changes.

The described techniques have design goals very similar to database cracking, namely automatic and adaptive index selection as well as incremental optimization of indexes focused on key ranges of interest in actual queries. The fundamental difference between the two approaches is the reliance on partitioning in database cracking and on merging in the new techniques. Well-known related algorithms are partitioning as used in quicksort and merging as used in external merge sort. The primary difference in the efficiency of the two techniques is that partitioning (using actual boundary keys of query ranges) is inherently limited to a partitioning fan-out of 2 or 3 whereas the merge fan-in is limited only by the available memory and thus can easily be in the 100s. With the number of partitioning or merging steps required to transform an initial index into the final, fully optimized index inversely proportional to the logarithm of the fan-out or fan-in, database cracking might move each data record 5-10 times more often than adaptive merging. Thus, adaptive merging requires less overall effort and adapts to changes in the query pattern much more rapidly than database cracking. Furthermore, like traditional B-trees and external merge sort, adaptive merging applied to partitioned B-trees is well suited to block-access storage.

Since partitioned B-trees are more similar to traditional B-trees than they are different, partitioned B-trees including adaptive merge strategies apply not only to B-trees with single-column search keys but also to multi-column "compound" B-trees as well as to B-trees on computed columns, e.g., B-trees on hash values or on space-filling curves such as UB-trees [B 97]. It seems that partitioned B-trees and adaptive merging applies in all cases in which traditional B-trees can be used.

Our plan for future work includes a more detailed experimental evaluation and research into incremental space reclamation by dropping obsolete or unused indexes. We will also investigate the trade-offs between index tuning prior to data loading (to save on in-database reorganization), traditional index tuning after observing a workload, and incremental index tuning using database cracking or adaptive merging. Finally, we recently presented a preview of this work to the team that developed database cracking at CWI, and agreed to a joint performance evaluation.

Acknowledgements

Martin Kersten and his research group invented database cracking, and the present research and design were sparked directly by his keynote at ICDE 2008. We particularly thank Stratos Idreos and Stefan Manegold for reviewing a draft of this paper, including our understanding of database cracking. Hans Zeller provided valuable feedback on our initial thoughts in May 2008. Barb Peters suggested several improvements in the text.

6 References

[B 97] Rudolf Bayer: The Universal B-tree for multidimensional indexing: general concepts. WWCA 1997: 198-209.

[BC 07] Bruno, N. and Chaudhuri, S. 2007. Physical design refinement: The 'merge-reduce' approach. *ACM Trans. Database Syst.* 32, 4 (Nov. 2007), 28.

[BM 72] Rudolf Bayer, Edward M. McCreight: Organization and maintenance of large ordered indices. Acta Inf. 1: 173-189 (1972).

[BM 93] Jon Louis Bentley, M. Douglas McIlroy: Engineering a sort function. Softw., Pract. Exper. 23(11): 1249-1265 (1993).

[BU 77] Rudolf Bayer, Karl Unterauer: Prefix B-trees. ACM TODS 2(1): 11-26 (1977).

[CN 07] Surajit Chaudhuri, Vivek R. Narasayya: Self-tuning database systems: A decade of progress. VLDB 2007: 3-14.

[FST 88] Sheldon J. Finkelstein, Mario Schkolnick, Paolo Tiberio: Physical database design for relational databases. ACM TODS 13(1): 91-128 (1988).

[G 03] Goetz Graefe: Sorting and indexing with partitioned B-trees. CIDR 2003.

[G 06] Goetz Graefe: Implementing sorting in database systems. ACM Comput. Surv. 38(3): (2006).

[G 07] Goetz Graefe: Hierarchical locking in B-tree indexes. BTW 2007: 18-42.

[G 07b] Goetz Graefe: The five-minute rule twenty years later, and how flash memory changes the rules. DaMoN 2007: 6.

[H 61] C. A. R. Hoare: Algorithm 64: Quicksort. Comm. ACM 4(7): 321 (1961).

[H 76] Theo Härder: Selecting an optimal set of secondary indices. ECI 1976: 146-160.

[IKM 07a] Stratos Idreos, Martin L. Kersten, Stefan Manegold: Database cracking. CIDR 2007: 68-78

[IKM 07b] Stratos Idreos, Martin L. Kersten, Stefan Manegold: Updating a cracked database. SIGMOD 2007: 413-424.

[IKN 08a] Milena Ivanova, Martin L. Kersten, Niels Nes: Self-organizing strategies for a column-store database. EDBT 2008: 157-168.

[IKN 08b] Milena Ivanova, Martin L. Kersten, Niels Nes: Adaptive segmentation for scientific databases. ICDE 2008: 1412-1414.

[KM 05] Martin L. Kersten, Stefan Manegold: Cracking the database store. CIDR 2005.

[LJB 95] Harry Leslie, Rohit Jain, Dave Birdsall, Hedieh Yaghmai: Efficient search of multi-dimensional B-trees. VLDB 1995: 710-719.

[NBC 95] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, David B. Lomet: AlphaSort: A Cache-Sensitive Parallel External Sort VLDB J. 4(4): 603-627 (1995).

[S 74] Michael Stonebraker: The choice of partial inversions and combined indices. International Journal of Computer and Information Sciences, 3(2), June 1974.

[S 89] Michael Stonebraker: The case for partial indexes. SIGMOD Record 18(4): 4-11 (1989).

[SS 95] Praveen Seshadri, Arun N. Swami: Generalized partial indexes. ICDE 1995: 420-427.