The VLDB Journal (2014) 23:303–328
DOI 10.1007/s00778-013-0345-7

SPECIAL ISSUE PAPER

Transactional support for adaptive indexing

Goetz Graefe · Felix Halim · Stratos Idreos · Harumi Kuno · Stefan Manegold · Bernhard Seeger

Received: 5 February 2013 / Revised: 27 October 2013 / Accepted: 30 October 2013 / Published online: 22 January 2014

Abstract Adaptive indexing initializes and optimizes indexes incrementally, as a side effect of query processing. The goal is to achieve the benefits of indexes while hiding or minimizing the costs of index creation. However, index-optimizing side effects seem to turn read-only queries into update transactions that might, for example, create lock contention. This paper studies concurrency control and recovery in the context of adaptive indexing. We show that the design and implementation of adaptive indexing rigorously separates index *structures* from index *contents*; this relaxes constraints and requirements during adaptive indexing compared to those of traditional index updates. Our design adapts to the fact that an adaptive index is refined continuously and exploits any concurrency opportunities in a dynamic way. A detailed experimental analysis demonstrates that (a) adaptive indexing maintains its adaptive properties even when running concurrent queries, (b) adaptive indexing can exploit the opportunity for parallelism due to concurrent queries, (c) the number of concurrency conflicts and any concurrency administration overheads follow an adaptive behavior, decreasing as the workload evolves and adapting to the workload needs.

Keywords Databases · Indexes · Adaptive indexing · Concurrency control · Robust query processing · Database cracking · Adaptive merging · Single-page failure

G. Graefe · H. Kuno (✉)
HP Labs, Palo Alto, CA, USA
e-mail: harumi.kuno@hp.com

G. Graefe
e-mail: goetz.graefe@hp.com

F. Halim
Google, Mountain View, CA, USA
e-mail: felix.halim@gmail.com

S. Idreos
Harvard University, Cambridge, MA, USA
e-mail: stratos@seas.harvard.edu

S. Manegold
CWI, Amsterdam, The Netherlands
e-mail: manegold@cwi.nl

B. Seeger
University of Marburg, Marburg, Germany
e-mail: seeger@informatik.uni-marburg.de

1 Introduction

This paper focuses on concurrency control for read-only queries in adaptive indexing. Adaptive indexing enables incremental index creation and optimization as automatic side effects of query execution. The adaptive mechanisms ensure that only tables, columns, and key ranges with actual query predicates are optimized [12,14,16,26–29]. The more often a key range is queried, the more its representation is optimized; conversely, columns that are not queried are not indexed, and indexes are not optimized in key ranges that are not queried. Prior research has introduced adaptive indexing in the forms of database cracking [20,26–28], adaptive merging [14,16] as well as hybrids [29] and benchmarking [12]. Past work focused on algorithms and data structures as well as on the benefits of adaptive indexing over more traditional index tuning and on workload robustness.

1.1 The problem: read queries become write queries

In adaptive indexing, queries executing scans or index lookups may invoke operations that incrementally refine the

database's physical design as side effects of query execution. Refinement operations construct and optimize index structures, causing logically "read-only" queries to update the database. This enables high-performance data loads followed immediately by efficient query processing, but raises the question whether the concurrency control required to support these updates incurs serious overhead and contention.

1.2 Index contents versus index structure

With regard to the concurrency control required to coordinate index updates, refining and optimizing an adaptive index during read queries is much simpler than updating a traditional index. Figure 1 illustrates the underlying intuition, comparing the incremental refinement of adaptive indexing to the explicit index updates involved in traditional indexing. The heights of each pair of boxes roughly illustrate the relative costs of various characteristics. Unlike traditional systems, in adaptive indexing, execution of read-only queries can trigger index updates and improves adaptively over time. On the other hand, the index changes caused by read queries impact only physical index structures, never logical index contents; thus, (a) concurrency can be governed using only *short-term in-memory latches* as opposed to transactional locks, and (b) the purely structural updates are optional and can be *skipped or pursued opportunistically*. These distinctions relax constraints and requirements with regard to concurrency control of adaptive indexing compared to those of traditional explicit index updates and enable new techniques for reducing the performance overhead of concurrency control during structural index updates.

1.3 Incremental granularity of locking

Another powerful characteristic of adaptive indexing is that the more an index is refined, the better its index structure supports concurrent execution by enabling a finer granularity of locking. That is to say, refinements to an index's structure enable updates to acquire increasingly precise locks. This effect is shown in the right part of Fig. 1, which illustrates the number of conflicts decreasing as the workload sequence evolves. Thus, as in query processing, concurrency control for adaptive indexing dynamically adapts to the running workload.

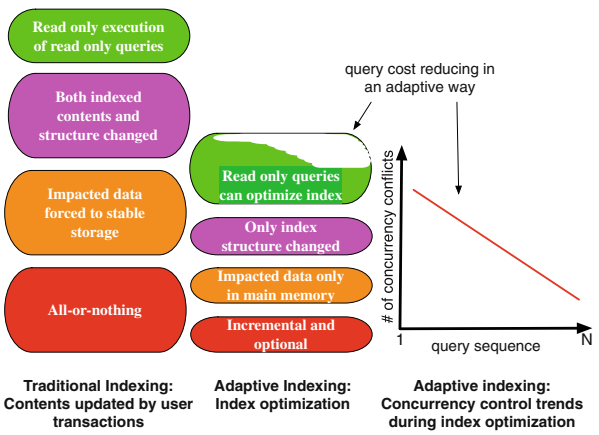


Fig. 1 Adaptive versus explicit indexing

1.4 Contributions

The current paper explores and proposes techniques for concurrency control, logging, and recovery that reduce the overhead imposed by adaptive indexing on read-only query execution to negligible levels. More specifically, we show the following:

- Adaptive indexing maintains its adaptive properties during the execution of concurrent queries.
- Concurrency conflicts adaptively decrease as adaptive indexing adjusts to the running workload.
- Adaptive indexing can exploit concurrent queries to increase parallelism.

In this paper, we focus on logically "read-only" queries that update (refine) the index only as a side effect of processing. We note that update algorithms for adaptive indexing have already been studied in [28] and that read–write conflicts in concurrent access can be resolved with the techniques reported here with minor modifications.

In the rest of the paper, we provide the necessary background on adaptive indexing; then, we discuss transactional support for state-of-the-art adaptive indexing approaches, and we present a detailed experimental analysis over a column-store system.

2 Prior work

Adaptive indexing changes the trade-off between load bandwidth (minimizing the number of optimized data structures created at data load time) and query performance (minimizing the effort spent on large scans and large, memory-intensive join operations). The defining characteristic of adaptive indexing is that indexes are created and refined *incrementally and continuously* as a side effect of query processing. This brings automatic adaptation of the physical design to the workload, but also introduces concurrency control issues during (adaptive) indexing. Although recent surveys on concurrency control and recovery [9,11] cover these topics for B-tree indexes and have influenced our research, to our knowledge, the present paper is the first to focus explicitly on how adaptive indexing mechanisms can support the transactional guarantees of the underlying database management system, without imposing undue overhead on the processing of read-only queries.

Table 1 Locks and latches

	index *Locks*	*Latches*
Separate…	User transactions	Threads
Protect…	Database contents	In-memory data structures
During…	Entire transactions	Critical sections
Modes…	Shared, exclusive update, intention, escrow, schema, etc.	Reads, writes, (perhaps) update
Deadlock…	Detection & resolution	Avoidance
…by…	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, "lock leveling"
Kept in…	Lock manager's hash table	Protected data structure

Before we discuss concurrency control and recovery methods for specific adaptive indexing techniques in Sects. 4 and 5, we first provide in this section the background necessary for understanding adaptive indexing mechanisms.

2.1 Transactional indexing techniques

2.1.1 Locks versus latches

The usual understanding of physical data independence focuses on tables and indexes. In addition, some data structures such as B-trees (and their variants) permit multiple representations for the same logical index contents. For example, a B-tree node may be compressed (shortened records) or compacted (no free space fragmentation), and it may contain "pseudo-deleted" "ghost" records (left by a deletion), etc. Similarly, boundary keys between nodes might be chosen by record count or by byte count, by length of the separator key [2], by desired "fill factor" (e.g., 90 % during database loading), etc.

The separation between logical index contents and physical data structure or representation affects the mechanisms used to enact their concurrency control. Locks separate transactions and protect logical contents, including the empty gaps between existing keys in serializable key range locking, whereas latches separate threads and protect data structures present in memory. Table 1, taken from [9], summarizes their differences. The crucial enabler is the separation of logical contents and physical structure.

2.1.2 User transactions and system transactions

The separation of logical contents and physical structure also affects the respective concurrency requirements of user versus system transactions. User transactions perform the database modifications requested by a user or application, whereas system transactions modify data structures without contents change. The purpose of system transactions is to enable efficient user transactions. For example, a user transaction might mark a record "pseudo-deleted" or a ghost, and a system transaction will later reclaim the space. System transactions are also very useful for splitting and merging nodes, load balancing between nodes, defragmentation, etc. Table 2, taken from [11], summarizes how system transactions differ from user transactions.

Table 2 User transaction and system transactions

	User transactions	System transactions
Invocation source	User requests	System-internal logic
Database effects	Logical database contents	Physical data structure
Data location	Database, buffer pool	In-memory page images
Parallelism	Multiple threads possible	Single thread
Invocation overhead	New thread	Same thread
Locks	Acquire and retain	Test for present locks
Commit overhead	Force log to stable storage	No forcing
Logging	Full "redo" and "undo"	Omit "undo" in cases
Recovery	Backward	Forward or backward

System transactions permit a number of optimizations during logging and recovery. First, although system transactions require a commit record just like user transactions, they do not need to force a commit record to stable storage. If a system transaction is not recovered after a system failure, only a representation change is lost, and if a subsequent user transaction relies on a prior system transaction, it will automatically, as a part of its own commit process, ensure that both commit records have been written to stable storage. Second, if a system transaction combines a log record (such as erasing a ghost record) with a commit record such that both occur in the same page of the recovery log, then there never can be a need to reverse this system transaction. Thus, system transactions can perform many clean-up tasks with less total log volume than that required for immediate clean-up by user transactions. Third, system transactions incomplete at the time of a system failure may recover by running to completion. Since system transactions do not affect logical database or index contents, the choice between roll-

back and completion cannot affect query results after system recovery.

2.1.3 Hierarchical and incremental locking

In traditional systems, the granularity of locking is fixed over time—multiple granularity choices may be available, but once chosen remains fixed for the system as a whole. For example, a workload made up of a multitude of concurrent small transactions might use fine-grained locking of individual keys, whereas coarser locks would enable large transactions to lock large ranges of keys efficiently without having to acquire a multitude of locks.

In order to reduce the number of locks required, hierarchical locking within an index can be employed. Hierarchical locking is a special case of multi-granularity locking that enables multitudes of small and large transactions to execute concurrently [8]. The key idea of hierarchical locking is that database objects must be locked according to their containment hierarchies. For example, a transaction that wanted to lock a leaf page in a B-tree index would first acquire a read lock on the table or view, then lock the index or index partition, and then finally lock the page. Multiple transactions could thus concurrently lock various leaves, and a subsequent large transaction could easily identify whether it can lock a partition without having to check each individual leaf's status.

Several designs for hierarchical locking in index exist, e.g., key range locking on separator keys within a B-tree index or on key prefixes of various lengths [8]. For example, if the artificial leading key field in a partitioned B-tree (see Sect. 2.2) is a 4-byte integer, Tandem's "generic lock" applied to the 4-byte prefix effectively locks an entire partition [8,36].

Hierarchical locking is limited to a predefined hierarchy of data structures, e.g., key, page, and index. The key idea of incremental locking is that the lock granularity can be changed dynamically, to adapt to the current workload. For example, given a workload that consists entirely of a multitude of small transactions in the morning and then shifts in the afternoon to eventually consist entirely of key range operations, an incremental locking system would automatically and dynamically shift from locking individual keys to locking key ranges. The partitions created by database cracking are naturally conducive to incremental locking; in that, the partitions created as a side effect of index refinement also represent sub-objects that can then be locked by subsequent operations.

2.1.4 Allocation-only logging

Allocation-only logging, sometimes also known as "minimal logging," means that only operations that impact space allocation are logged. Updates to page contents are not logged; instead, page contents are forced to storage (traditionally, to disk) prior to transaction commit ("force" commit policy). These forced, non-logged, pages are included in the next transaction backup. Many commercial database systems use allocation-only logging in order to reduce log volume by orders of magnitude, compared to a naive logging technique for index operations. Allocation-only logging applies to index operations, e.g., creating a new secondary index, and to page operations, e.g., load balancing among two sibling nodes in a B-tree.

After a page operation moves records between two-page images in the buffer pool, the affected pages are written to storage in such a sequence that the destination page is written before the source page. This ensures that no records can be lost in a system failure.

For an index operation, rollback simply releases the new index pages as free space. Commit of an index operation with allocation-only logging forces all new index pages to permanent storage [10]. Moreover, the next backup of the recovery log should include these pages; otherwise, a sequence of log backups may not permit correct recovery of a media failure.

2.2 Partitioned B-trees

Offering a simple and efficient mechanism for partitioning the contents of a single B-tree, partitioned B-trees provides an ideal foundation for adaptive indexing in the form of adaptive merging.

There are three differences between a partitioned B-tree and a B-tree partitioned in a traditional parallel database management system. First, and most importantly, a partitioned B-tree is a single B-tree, whereas a traditional partitioned index employs a B-tree per partition. In fact, each individual B-tree in a traditional partitioning scheme might actually be a partitioned B-tree in order to support efficient index creation and incremental loading. Second, partitions in a traditional partitioning scheme are listed in the database catalogs, whereas a partitioned B-tree contains multiple partitions simply by means of distinct values in the artificial leading key field. Third, partitions in a traditional partitioning scheme require catalog updates with the attendant concurrency control protocols, e.g., exclusive locks on data and metadata of a table, whereas partitions in a partitioned B-tree appear and disappear simply by insertion and deletion of records with appropriate values in the artificial leading key field.

For example, Fig. 2 illustrates data that have been loaded into a partitioned B-tree. Each partition holds as much data as could be sorted in-memory. The advantage of this approach is that the data could be loaded into the B-tree without requiring a full sort. The disadvantage is that queries against the tree should now account for the multiple partitions.

In a partitioned B-tree, each initial run forms its own partition. Similarly, when partitions are merged, the results may

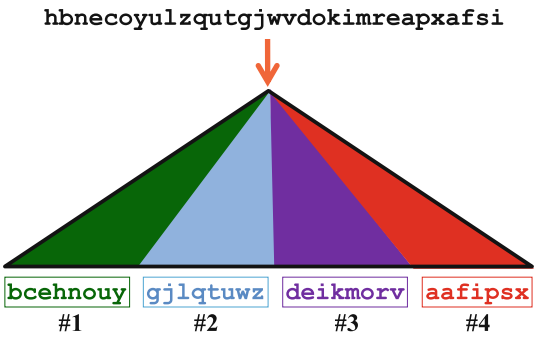


Fig. 2 Data loaded into a partitioned B-tree

form a new partition. Partition contents are managed using a table of content data structure. Earlier work has proposed alternative data structures for the table of contents [15].

2.3 Online index operations

For a detailed discussion of online index operations, readers are referred to [10], which the following text briefly summarizes. Online index functionality enables concurrent queries and updates during index maintenance tasks, e.g., creation of a new secondary index. Online index creation permits concurrent transactions to also update the table, including insertions and deletions, with the updates correctly applied to the index before index creation commits. There are two principal designs for online index operations. In the "no side file" approach, concurrent updates are applied to the structure still being built [10]. Alternatively, concurrent updates are captured elsewhere in a "side file" and applied after main index creation activity is complete [34]. The recovery log may serve as the "side file." Srinivasan and Carey further divide "side file" online algorithms for index creation according to whether concurrent updates are captured in a list or in an index [39]. (Srinivasan and Carey do not consider capturing updates in the recovery log.)

The "side file" design lets the index creation proceed without regard to concurrent updates, but requires at least one "catch-up" phase that in turn requires either quiescent concurrent update activity or a race between capturing updates and applying them to the new index [10]. The "no side file" design requires the index creation process work around records in the future index inserted by concurrent update transactions. For example, concurrent update transactions may need to delete a key in a key range not yet inserted by the index creation process because index creation is still being sorting records to be inserted into the new index. Such deletions can be represented by a negative or "anti-matter" record [10]. When the index creation process encounters an anti-matter record, the corresponding record is suppressed and not inserted into the new index, after which the anti-matter record, having served its function, can be removed from the B-tree.

2.4 Adaptive indexing

Many prior index tuning and management approaches focus on optimizing decisions related to the management of full index structures that cover all key ranges [3–6,21,24,37]. Some approaches recognize that some data items are more heavily queried than others and support partial indexes [35, 40], while others recognize that not all decisions about index selection can be taken up front and provide online index operations [3,37]. For these prior approaches, explicitly creating and refining index structures using independent operations does not impose additional concurrency overhead upon the processing of read-only queries. Full or partial indexes are created either up front or periodically, interleaving query execution. The interested reader may find a detailed description and study of these basic adaptive indexing techniques in [25].

2.4.1 Database cracking

"Database cracking" pioneered focused, incremental, automatic optimization of the representation of a data collection —the more frequently a key range is queried, the more its representation is optimized for future queries [20,26–28,30]. As its name suggests, database cracking splits an array of values into increasingly refined partitions. Developed in the context of the column-store database MonetDB, to work on byte-addressable in-memory data, one can think of it as an incremental quicksort where each query may result in a partitioning step. Index optimization is entirely automatic and occurs as a side effect of queries over key ranges not yet fully optimized.

For example, Fig. 3 shows data being loaded directly, without sorting, into an unsorted array. A read-only query on the range "d–i" then arrives. As a side effect of answering that query, the array is split into three partitions: (1) keys before "d"; (2) keys that fall between "d" and "i"; and (3) keys after

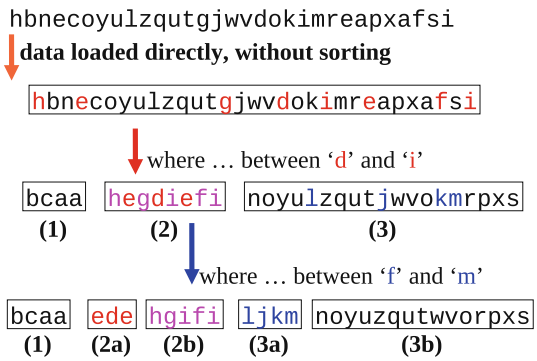


Fig. 3 Database cracking

"i." Then, a new query with range boundaries "f" and "m" is processed. The values in partition (1) can be ignored, but partitions (2) and (3) are further cracked on keys "f" and "m," respectively. Subsequent queries continue to partition these key ranges until the structures have been optimized for the current workload.

2.4.2 Adaptive merging

Inspired by database cracking, and developed in the context of page-access storage, "adaptive merging" also refines index structures during query processing [14,16]. While database cracking resembles an incremental quicksort, with each query resulting in at most two partitioning steps, adaptive merging resembles an incremental external merge sort, but with the merge effort expended only on demand and with both run generation and merging realized as side effects of query execution. In adaptive merging, the first query with a predicate against a given column produces sorted runs. Each subsequent query against that column then applies at most one additional merge step to each record in the desired key range. All records in other key ranges are left in their initial or current places. As with database cracking, this merge logic takes place as a side effect of query execution.

In adaptive merging using partitioned B-trees, a run in the sort logic equals a partition in the B-tree. The first step creates as many new partitions as required to capture all new index entries in runs. Subsequent steps scan key ranges in the initial partitions and merge them into the final partition. Earlier work has proposed alternative data structures for the table of contents [15]. For very large tables and very limited merge fan-in, multiple merge levels may be required. Each complete merge level moves each record once; in other words, in many cases, each record participates in only one merge step. In all cases, records are not duplicated between partitions; each record logically resides in only one partition at a time.

In its first step, adaptive merging scans the database table (or other data source) and creates initial partitions using traditional techniques for run generation, i.e., read–sort–write cycles using quicksort or continuous replacement selection using a priority queue. For example, Fig. 4 shows an initial read-only query that creates equally sized partitions and sorts them in-memory to produce four sorted runs. In subsequent steps, it merges specific key ranges using traditional techniques for range queries and for external merge sort. Continuing the previous example, while a second query with range boundaries "d" and "i" is processed, relevant values would be retrieved (via index lookup because the runs are sorted) and merged out of the runs and into a "final" partition. Similarly, results from a third query with range boundaries "f" and "m" are merged out of the runs and into the final partition. Subsequent queries continue to merge results from the runs until

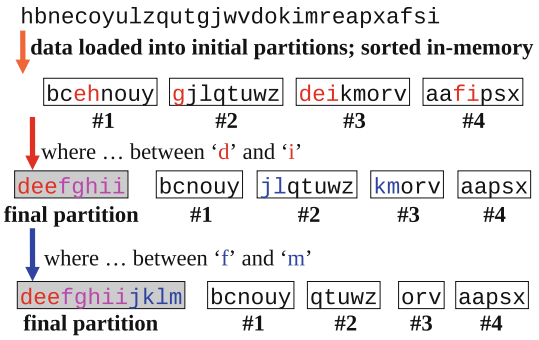


Fig. 4 Adaptive merging

the "final" partition has been fully optimized for the current workload.

2.4.3 Hybrid adaptive indexing

Database cracking and adaptive merging have distinct strengths; our adaptive indexing "hybrid" approach brings together both sets of strengths in the context of an in-memory column store [29]. Each step of database cracking is like a single step in a quicksort, whereas the first step of adaptive merging creates runs, in which subsequent steps merge. Thus, database cracking enjoys a low initialization cost, but converges relatively slowly, whereas adaptive merging has a relatively high initialization cost but converges quickly to an optimally refined index.

Our hybrid adaptive indexing algorithms apply different refinement strategies to initial versus final partitions, exploiting the insight that in adaptive merging, once a given range of data has moved out of initial partitions and into final partitions, the initial partitions will never be accessed again for data in that range. A final partition, on the other hand, is searched by every query, either because it contains the results or because results are moved into it. Therefore, effort that refines an initial partition is much less likely to "pay-off" than the same effort invested in refining a final partition.

The hybrid algorithms combine the advantages of adaptive merging and database cracking while avoiding their disadvantages: *fast convergence, but without the burden of fully sorting the initial partitions.* For example, Fig. 5 shows an initial read-only query that creates four equally sized unsorted initial partitions. While a second query with range boundaries "d" and "i" is processed, each initial partition is cracked on the query's boundaries, and the requested values are merged out of the initial partitions and into a sorted "final" partition. Similarly, from a third query with range boundaries "f" and "m," the initial partitions that hold relevant values are cracked, and the result values are merged out of the initial partitions and into the final partition. Subsequent queries continue to crack initial partitions and merge results from them

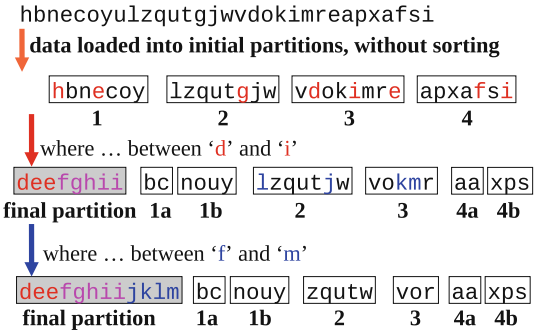


Fig. 5 Hybrid "crack-sort" adaptive indexing

until the "final" partition has been fully optimized for the current workload.

2.4.4 Soft indexes

"Soft indexes" automatically and autonomously manage indexes in response to a workload [32]. Like the monitor-and-tune approaches, this approach continually collects statistics for recommended indexes and periodically and repeatedly automatically solves the index selection problem. Unlike typical monitor-and-tune approaches and like adaptive indexing approaches, [32] then generates (or drops) the recommended indexes as a part of query processing. Unlike adaptive indexing approaches such as database cracking and adaptive merging, however, neither index recommendation nor creation is incremental; explicit statistics are kept, and each recommended index is created and optimized to completion (although the command might be deferred). Although we recognize soft indexes as a kin to database cracking and adaptive merging, in the remainder of this paper, we focus upon the latter, i.e., incremental and adaptive indexing methods.

2.5 B^{link}-trees and foster B-trees

Adaptive indexing in the form of adaptive merging, which is discussed in Sect. 4, potentially physically restructures a B-tree's nodes with every query. In order to reduce the likelihood of latch contention, particularly in the context of multi-core backed by fast persistent memories, we recommend the use of Foster B-tree mechanisms when implementing adaptive merging in a B-tree.

The classic latching technique would advocate that the root-to-leaf pass for an insertion retains an exclusive latch until it passes through a "safe" node that has sufficient free space for a local insertion in case a child node must split and post a new branch key (and thus cannot require a split) [1]. This potentially results in an exclusive latch on every node involved in the root-to-leaf search for the insertion point.

Improving upon B^{link}-trees,[31], Foster B-trees require only two latches at a time during a structural modification. Their design introduces left-to-right sibling pointers in addition to parent-to-child pointers. Each level of the B-tree data structure forms a singly linked list. When a node overflows and a new node is required, the overflowing node holds a pointer to the new node together with a key value that separates key values retained in the old node and those moved to the new node. Because the parent node does not participate in the split operation, two latches suffice to protect the overflowing node and the new node. Soon thereafter, the pointer and the branch key are copied to the parent node, which requires latches only on the sibling and the parent nodes. If thereupon the parent node overflows, the same techniques are applied there.

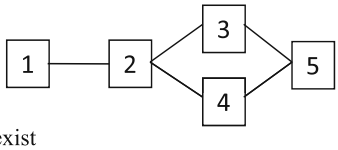
Like B^{link}-trees,Foster B-tree relationships avoid the need for root-to-leaf exclusive latches for insertions. Foster B-trees have three unique characteristics that help them avoid latch contention:

1. Every node in the tree has only a (single) incoming pointer at all times.
2. A node may temporarily provide the single incoming pointer to a sibling node, a relationship described using the terms "foster parent" and "foster child." Like B^{link}-trees before them [31], Foster B-trees require only *two* latches at a time during a structural modification. For example, when a node is first split, because the parent node does not participate in the split operation, only two latches are needed—one to protect the overflowing node and one to protect the new node.
3. Any structural change (e.g., the split of an overflowing node, the insertion of a new node, or the deletion of an underflown node) can be represented as a sequence of *three* independent incremental operations, none of which requires more than two latches at any time. For example, the intermediate state during a leaf split is transient and resolved quickly after it has been created, but it may persist long enough to be observed by other threads or other transactions. Resolving it means moving pointer and branch key from the formerly overflowing sibling node to the parent.

We refer interested readers who would appreciate a more detailed discussion to a prior publication [13].

3 Approach

While our prior work on adaptive indexing [12,14,16,20,26–30] has focused on data structures and algorithms, transactional guarantees are also required for integration of new techniques into a database management system. We are informed by recent surveys on concurrency control and recovery [9,11] that cover these topics for B-tree indexes. The



States

1. Index does not exist
2. Index is in the catalogs but does not contain any records
3. Partial index in a traditional online index operation
4. Partially optimized index in adaptive indexing
5. Finished index – fully populated and fully optimized

Fig. 6 Traditional online versus adaptive indexing

transactional ACID guarantees include (failure) atomicity, consistency, isolation (synchronization atomicity), and durability. Database systems usually implement recovery (failure atomicity, durability) by write-ahead logging and concurrency control (synchronization atomicity) by locking and latching.

Adaptive indexing introduces new potential states to the index life cycle. The diagram in Fig. 6 compares the relationship between index states in traditional online index operations versus in adaptive indexing. State 2 is invisible and of very short duration in most systems. An exception is Microsoft SQL Server, where this state is called a "disabled index." In State 3, the index is partially populated, i.e., it contains fewer index entries than there are rows in the underlying table, but the index is fully optimized. That is, those key ranges already in the index are in their final position within the index. Table and index can be updated (that is the "online" aspect of traditional online index creation), but the index cannot be used for search during query processing.

Adaptive indexing refinements take place in State 4, where the index is fully populated but the index structure is not fully optimized. In other words, all index entries exist but not yet in their final position. In this state, the fully populated, partially optimized index is available for both read-only query processing and read-write update processing, whereas the partially populated, fully optimized partial index in online index creation requires effort in all updates but does not contribute to read-only query processing. Optimization of the index is left to future query execution and will affect only those index entries relevant to actual queries, i.e., key ranges in actual predicates. Optimization of other key ranges is deferred until relevant queries are encountered, possibly indefinitely. A single user transaction might encounter multiple such cases, e.g., querying and updating multiple key values in a single index, which is being optimized by adaptive indexing techniques.

This section discusses at a general level how to provide transactional guarantees to concurrent queries and refinement operations against an index that is in State 4. We consider concurrency control mechanisms that ensure that concurrent transactions can operate in isolation, how to provide durability while minimizing the performance overhead of logging, and how to recover from failures without losing prior refinement efforts or sacrificing failure atomicity. For each topic, we discuss general techniques that facilitate the task in the context of adaptive indexing. Next, in Sects. 4 and 5, we demonstrate how to apply those techniques to adaptive merging and database cracking, respectively.

Fundamentally, two factors mitigate the potential complexity and overhead that adaptive indexing incurs when executing queries that are logically "read-only" but which refine index data structures. First, with read-only queries, adaptive indexing performs only *structural* modifications to the physical representation of the index, leaving the logical *contents* of the index unmodified. This separation between user data and system state is very powerful and gives the system transactions of adaptive indexing independence from user transactions, even if they run within the same execution thread. For example, if a user transaction rolls back for some reason, there is no need to reverse its index optimization already achieved. More subtly, the user transaction (and its query) might run in a low transaction isolation level, e.g., read committed, whereas the index optimization must achieve complete correctness and synchronization atomicity with respect to all other transactions active in the system.

Second, the adaptation of index data structures to conform to the current workload enables the automatic and dynamic adaptation of the lock granularity of locks needed to coordinate structural changes. That is, as the workload progresses and the physical data structures become increasingly refined, not only do structural changes become less likely, but also the objects locked by refinement operations become increasingly finer-grained, reducing the likelihood of contention.

3.1 Concurrency control

The following focuses on a single-threaded query with index optimization and on concurrency control with respect to other queries. This scenario is particularly relevant with regard to State 4 as shown in Fig. 6, where an index has been created and added to the catalogs, but the index has not yet been fully optimized for this workload, so queries may still result in updates to index structures.

3.1.1 Concurrency control by latching

Since index optimization affects only index structure, not logical index contents, the thread and system transaction performing the index reorganization may rely entirely on latches. There is no need for acquisition of any locks, although it is required to verify that no concurrent user transaction holds conflicting locks. The latches (on index pages) are retained during a quick burst of reorganization activity; as in standard system designs, user transactions cannot request locks

on a page or on key values within an index page without first acquiring the latch on the page.

3.1.2 Conflict avoidance

Index reorganization in adaptive indexing is *optional*. Adaptive indexing treats each read query as an opportunity to improve the physical design. All actions are optional, as adaptive indexing inherently operates on incomplete, not fully optimized indexes. In other words, if an individual query fails to optimize the index, some other query will do so soon thereafter if necessary—and the bigger the potential impact of the refinement action, the more likely that it will eventually take place. Thus, if a query intends to optimize an index but finds that some concurrent user transaction holds conflicting locks, the query can simply forgo the index optimization.

3.1.3 Early termination

Some forms of adaptive indexing can terminate an optimization step at any time yet leave behind a consistent and searchable index, which subsequent queries and their side effects may optimize further. Thus, if a user transaction attempts to access pages latched by an active system transaction performing index optimization, the system transaction may terminate instantly, release its exclusive latches, or downgrade them to shared latches, permitting the concurrent user query to proceed.

Concurrency contention is one of two possible cases for early termination of system transactions engaged in index optimization. The other case is discussed below in the context of recovery.

3.2 Implicit multi-version concurrency control

Finally, adaptive indexing lends itself naturally to multi-version concurrency control. Because index structures are independent from contents, two transactions may each operate upon their own copy of a contended index structure, which may be assigned version numbers.

3.3 Logging

As adaptive indexing creates and optimizes (typically) optional indexes, and since this process might move each data item multiple times, minimal overhead is crucial. In other words, both creation of the initial index (State 2 in Fig. 6) and each optimization step (State 4) should log at most key ranges, page allocation actions, etc. but not the contents of records and of index pages. The most closely related technique in traditional index creation is "allocation-only logging," which, as described in Sect. 2.1.4, saves most of the naive logging effort (by logging allocation of pages but not page contents) and which we adapt to adaptive indexing.

3.3.1 Retaining old pages

The separation of logical contents versus physical structure again permits multiple optimizations. User transactions and system transactions have different requirements, not only with respect to failure atomicity but also with respect to durability. In other words, until a subsequent user transaction updates the logical index contents and that user transaction's commit processing forces all prior log records to stable storage, a system transaction can be recovered by "undo" (compensation). This can be as simple as dropping modified (dirty) pages in the buffer pool and thus retaining the data pages valid prior to a reorganization step in an index.

3.3.2 Small system transactions

In online index creation, which focuses on concurrent queries and user updates, one transaction builds the index and another transaction, executing in another thread, applies updates [34]. In adaptive indexing, structural index optimization runs within the same thread as the main user query but individual transactions, user and system transaction, separate user query and index optimization.

A further technique exploiting system transactions is the following. Instead of a single large system transaction, many small system transactions can accomplish the same effect. Low invocation and commit overheads limit the cost of a large number of transactions. Rather than supporting termination of system transactions due to newly arrived conflicting transactions, the system may let active system transactions finish and merely not start new ones.

3.3.3 Dedicated update partitions

In both adaptive indexing methods, database cracking and adaptive merging, concurrent user transactions may apply their updates to storage locations dedicated to the purpose. Subsequent reorganization steps can work recently added records into the main index including deletions initially represented by insertion of "anti-matter" records. The user transactions log their initial insertions in the normal way; subsequent reorganization can benefit from the log optimizations above.

3.4 Recovery

3.4.1 Starting over

Again, as adaptive indexing creates and optimizes (typically) optional indexes, one possible recovery after a system or

media failure is to drop the index. In fact, one could argue that this approach may reduce the restart and recovery time. The approach is conceivable even as recovery from a failed transaction. However, even if most system transactions for index optimization succeed, this seems a rather drastic measure after failure of a single transaction as it loses all past index optimization effort.

3.4.2 Leveraging prior query patterns

Even when the index structure is corrupted or lost, information about prior query patterns may still be intact. If that information is still available, then one recovery strategy is to deallocate the index, but pro-actively start optimizing the index based on prior query patterns. For example, a query log or some equivalent data structure could guide initial optimization of the index structure, possibly by re-invoking earlier queries for the benefits of their side effects.

3.4.3 Selective undo and redo

The traditional recovery technique performs complete "undo" or "redo" recovery. If a reorganization step is committed and the recovery logic finds the commit record in the recovery log, a subsequent user transaction might depend on the reorganization's effects. Therefore, only "redo" recovery is acceptable. This is subject to the rules for "restricted repeating of history" [11].

Without a commit record in the recovery log, the recovery logic is free to choose between "undo" (compensation) and "redo" recovery, because index reorganization affects only the index structure, not index contents. For example, if the reorganization step is complete with only the commit record missing, e.g., because system transactions do not force their commit records to stable storage, then recovery can be much more efficient if it completes the reorganization by simply adding the missing commit record.

3.4.4 Early termination

In fact, the recovery logic may even commit a partially complete reorganization step. The only requirement is that the index is in a consistent state after recovery, i.e., the index contains precisely the correct index entries exactly once. Thus, early termination of a reorganization step can aid not only concurrency control but also efficient recovery.

3.5 Summary

In summary, efficient concurrency control and recovery requires strict separation of logical index contents and physical index structure. Reorganization of an index, whether database cracking or adaptive merging, does not affect its logical contents. Thus, index optimization can avoid conflicts with user transactions and locks. Instead, it can rely on system transactions, latches, and many small transactions with low overheads for invocation and commit processing. These system transactions must respect existing locks held by user transactions, but the system transactions have no need to acquire and retain locks.

Moreover, some optimizations are specific to content-neutral index operations. In case of concurrency contention, a system transaction may simply stop, commit work already completed, and defer further planned work to a subsequent system transaction. In case of a system failure during index optimization, recovery may choose to "redo" or "undo" a system transaction and its index reorganization, and it may even choose partial "redo" and partial "undo" as long as the recovery remains contents-neutral like the original system transaction. We are currently considering whether these properties are more general, e.g., apply to all system transactions, not only index optimization in adaptive indexing techniques.

4 Adaptive indexing with B-trees

The recommended data structure for adaptive merging is a partitioned B-tree, preferably one that also employs Foster B-tree mechanisms. Therefore, many techniques designed for B-tree indexes can be used, not only with respect to data structures, storage management, etc., but also with respect to concurrency control, logging, and recovery.

4.1 Transactions and partitioned B-trees

Transactional guarantees in adaptive merging rely on combining partitioned B-trees with the techniques outlined in Sect. 3. However, no special latching mechanisms are needed. Recall that a partitioned B-tree is implemented using a single B-tree and that conceptually partition identifiers are simply artificial keys prepended to the keys within a given partition. Merging records from one partition into another is thus simply a matter of updating the keys associated with those records, substituting one partition's identifier for another's.

For example, concurrency control and recovery can rely on the techniques explored in earlier research and development, e.g., [9,11,19,22,33]. In the following, we point out specific techniques for B-trees and adaptive merging with particularly low overhead and low contention.

In adaptive indexing, no individual query and its execution rely on a specific earlier query and its optimization of an index structure or even completion of the B-tree index. Thus, several techniques for efficient concurrency control and recovery rely on interpreting any requested index optimization as optional. For even more flexibility, input and output pages in merge steps can enable multi-version con-

currency control to separate read-only queries and queries with index optimization as side effect.

Adaptive merging relies on a form of differential files [38] for high update rates. During a single load operation, multiple new partitions might be created in a partitioned B-tree. Typically, the size of each new partition is equal to (or twice) the size of the memory available for sorting arriving records. Concurrent update transactions may apply their updates and deletions immediately in place or defer them by insertion of "anti-matter" (deletion markers), which are used routinely in online index creation and in incremental maintenance of materialized views [10]. Insertions may be collected in new partitions within the partitioned B-tree or they may be applied to an existing partition. We recommend new partitions for key ranges not yet fully optimized, i.e., key ranges with records still scattered in multiple partitions, and immediate maintenance of the existing, final partition for fully optimized key ranges.

4.2 Concurrency control

At each query, adaptive merging potentially optimizes index structures by moving records from one partition to another, raising two challenges. First, concurrent queries upon overlapping key ranges potentially increase the possibility of lock contention. Second, even if two optimization actions do not contend for logical locks, they could potentially contend for physical latches. We bypass the first challenge by observing that index optimization actions impact only the database's organization, as opposed to its contents, and thus may be skipped. We address the second challenge by leveraging the Foster B-tree mechanisms described in Sect. 2.4.4. We further discuss both of these challenges below.

4.2.1 Locking

Early research on concurrency control in B-trees failed to separate short-term protection of the data structure versus long-term protection of B-tree contents. The distinctions of contents versus representation, user transactions versus system transactions (outlined in Table 2), locks versus latches (sketched in Table 1), etc., are now standard in sophisticated B-tree implementations. Key range locking for leaf keys is also standard, and key range locking for separator keys explicitly relies on the structure of B-trees (page locking). Thus, all these techniques immediately apply to adaptive merging implemented with partitioned B-trees.

That is to say, refinement of index structures is effected by system transactions, which test for locks, but do not acquire or retain them. Thus, the transactions that realize adaptive merging's index refinements will never create lock contention for user transactions. Just as important, as described in Sect. 3, a system transaction that encounters an exclusive lock on a page it intends to modify can abort without any impact on database contents because index refinement is an optional activity.

Furthermore, the partitioned B-tree recommended for adaptive merging is naturally conducive to concurrency control. A partitioned B-tree is a valid B-tree index, with respect to both contents and representation, independent of the merge steps completed. The original partitioned B-trees [7] exploited this property in various ways. It can also be exploited for concurrency control in adaptive merging. In particular, concurrency control conflicts should, when possible, be resolved by instantly committing an active merge step and its result.

Finally, merge steps take records from many existing B-tree pages and write new pages in a new B-tree partition. Although the record-to-be-merged may need be merged with each other, they will at worst be interleaved with the contents of the "final" partition. These separate sets of pages readily enable a limited form of multi-version concurrency control, with shared access to the old pages and exclusive access to the new pages until they are committed.

Therefore, if one transaction attempts to merge index entries but finds (e.g., by latch conflicts) that the key range of interest is being merged already by another transaction, it should simply scan the key range, forgo any side effect, and return the desired query result.

4.2.2 Latching

As described in Sect. 2.4.4, traditional latching techniques require two adaptive merging optimization actions that write to separate pages within the "final" partition to each obtain and retain exclusive latches for each node encountered along the insertion root-to-leaf pass until it passes through a "safe" node. The two optimization actions could thus find themselves contending for an exclusive latch on interior nodes, even though they will insert content into different leaf nodes.

Adaptive merging thus particularly benefits from the at most two exclusive latches required by Foster B-tree operations. For example, Fig. 7 shows a portion of a B-tree corresponding to part of an adaptive merging final merge partition. Two independent yet concurrent optimization actions each want to merge records into separate pages. The records to be merged along with their target destinations are outlined in blue (left) and in purple (right), indicating that they are associated with separate system transactions. The blue (left) transaction results in records that lie in the range between 20 and 39. Previous queries have partially refined this range, and thus the results of the current query must be interleaved with preexisting records that were added to the final partition by prior queries. For the sake of simplicity, these records are represented in the figure by the keys '25', '30', '40', etc., but in realistic situations (too complex to draw here) likely

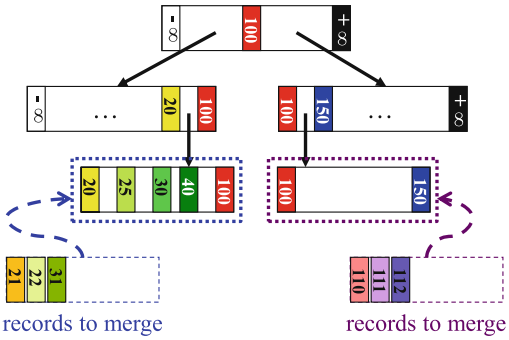


Fig. 7 Records being merged into the final index partition by concurrent system transactions (*blue* and *purple*) (color figure online)

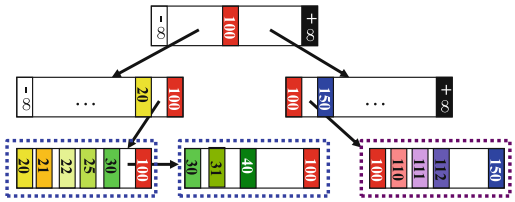


Fig. 8 One system transaction (*blue*) holds two latches; the other (*purple*), holds one (color figure online)

would represent ranges of keys. With foster child relationships, at this point, only the target nodes need to be exclusively latched, indicated by the dotted blue and purple lines.

Figure 8 shows an intermediate state, where the merged records have been inserted, and latches are just being released. The blue- and purple-dotted boxes indicate the node latches that are about to be released by the two system transactions. Note that the (left) blue transaction resulted in the split of a leaf node being split and the creation of a foster child, whereas the (right) purple transaction relation was able to complete the merge without any node splits.

Algorithm 1 sketches the process of how latches are acquired and released by a system transaction such as the ones illustrated in Figs. 7 and 8 that logically merges records from run partitions into a leaf node of the final partition.

4.3 Logging

With the focus here on adaptive indexing, we assume that index updates initiated by users and applications employ standard logging. Automatic index creation and optimization, on the other hand, must be prevented from producing excessive log volumes, just like traditional index utilities.

The goal, therefore, is adaptive merging with allocation-only logging yet with recovery comparable to fully logged index operations. In the following, we assume allocation-only logging for both run generation and all incremental steps in adaptive merging, i.e., both for saving future index entries in multiple runs (B-tree partitions) and for merging entire

Algorithm 1 MergeResultsIntoLeaf(recordToMerge)
Insert a record-to-be-merged into a leaf node while protecting the leaf node with latches. Although it is now shown, if any latch attempt fails, the merge attempt aborts.

```
1: // Find the leaf node where the record-to-be-merged belongs.
2: readLatch(RootNode );
3: node = RootNode.lookup(recordToMerge);
4: readLatch(node );
5: unlatch(Root);

6: while ! node.isLeaf() do
7:    nextNode = RootNode.lookup(recordToMerge);
8:    readLatch(nextNode );
9:    unlatch(node);
10:   node = nextNode;
11: writeLatch(node);
12: // Insert the record.
13: node.insert(recordToMerge);
14: // Split the leaf if necessary.
15: if node.size() >= splitThreshold then
16:    node.split();
```

partitions or limited key ranges during index optimization. While allocation-only logging reduces the logging overhead incurred by adaptive merging, the difficulty is in ensuring correct and complete recovery from all failure.

4.4 Recovery

Recovery of the index contents from log records is not possible after allocation-only logging—the new index's contents must be completely re-created. Even so, adaptive indexing offers better recovery techniques than simply re-creating an index by running the entire index creation logic from start to finish. Below, Sect. 4.4.1 describes how the replacement index may be created incrementally, e.g., broken into initial extraction of future index entries and incremental optimization of the index structure. Next, Sect. 4.4.2, proposes a second, novel, alternative, which avoids having to completely re-create the entire index by exploiting the single-page recovery of all pages in the index, despite allocation-only logging.

4.4.1 Incremental index re-creation

For any optional index, a possible recovery technique is to start over with an entirely new index. Note that adaptive indexing offers more options for recovery techniques than just re-creating an index, running the entire index creation logic from start to finish: the replacement index may be created incrementally, e.g., broken into initial extraction of future index entries and incremental optimization of the index structure. Moreover, these individual steps can again be side effects of query execution. Finally, the same techniques for concurrency control, logging, and recovery apply during a first and any subsequent attempt to build an index.

For example, adaptive merging indexes can benefit from a technique for recovering B-tree indexes that are created by sorting the future index entries. The technique exploits knowledge of prior query patterns, by permitting preferential treatment of certain key ranges during run generation in order to reduce the merge effort for those key ranges only. It is adapted from a sorting technique that produces the lowest key values with lower merge effort and even no merge effort at all. Specifically, memory is divided during run generation into two workspaces for "low" and "other" key values. The fraction of memory for "low" key values is larger than the fraction of "low" key values in the input data.

One effect of this division is that runs are smaller and more numerous; the other effect is that few runs contain the lowest keys. For example, if 10 % of the records fall into the "low" key range and 50 % of memory is used for each workspace, run generation produces almost twice as many runs but only about 10 % of those runs contain "low" keys. Thus, if the merge fan-in is limited, merge effort for the "low" keys is reduced. If the fraction of "low" keys is very small, merging might be avoided entirely. If the sort operation is used to prepare B-tree creation, the "low" keys can be any favored key range instead. If the query pattern or interesting key ranges are known from prior queries, re-creation of an index during recovery can optimize the sort algorithm accordingly.

The flexibility of partitioned B-trees readily permits early termination during recovery. In other words, recovery may undo an incomplete merge step, redo and complete it, or redo it for some of the key range and undo it for the remainder. The last option is particularly valuable if it permits the recovery process to leave most on disk pages in their current state, thus speeding up the entire recovery process.

4.4.2 Single-page logical recovery

Whereas the preceding subsection outlined how to recover an entire adaptive index incrementally, we delineate here a simple strategy for recovering from partial failures—for example, if just a certain key range of the adaptive index is lost. This strategy extends our earlier research on single-page failures, their detection, and recovery. The recommended strategies apply logical recovery, i.e., re-deriving index contents, rather than physical recovery, i.e., copying lost index contents from the log [18,41,42]. Strategies differ for initial index partitions and for intermediate or final index partitions. The former are derived from a table's primary storage container, e.g., a clustered index or index-organized table, and recovery re-derives index contents from the same source. Merge output is obtained from earlier partitions (runs) and recovery re-derives index contents by invoking the appropriate merge logic. Moreover, recovery strategies differ by the scope of the data loss, which may affect an entire index partition or merely an individual page (or a few individual pages).

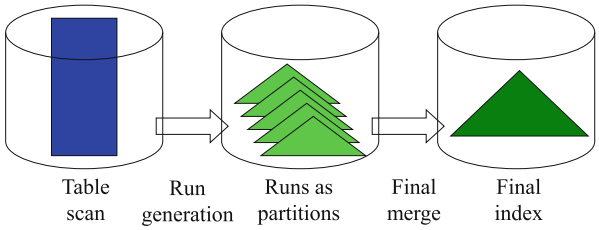


Fig. 9 Traditional recovery

If a traditional system suffers a system or media failure during index creation, recovery of the newly created secondary index relies on repeating the entire index operation, i.e., it reaches back to the base table. Recovery of the index contents from log records is not possible after allocation-only logging—the new index's contents must be completely re-created.

Figure 9 schematically illustrates the problem of the traditional recovery process. Runs in the external merge sort are shown as partitions of a B-tree; the final merge step creates the desired B-tree index; and recovery after failure reaches back to the base table, i.e., repeats the entire effort of index creation.

The preceding section indicates a better alternative, namely re-creating an index incrementally over time, preferably as side effect of query execution. But better yet, incremental recovery pertains not only to the traditional failure modes (i.e., transaction failure, media failure, system failure) but also to a fourth failure mode added only recently, single-page failure [17]. Such failures may be due to defective B-tree code, lower software levels within the database management system, or software and hardware serving the database management system, e.g., storage-area networking and snapshot file systems.

The following discussion focuses on failure and recovery of individual leaf pages or groups of leaf pages in B-tree indexes. For efficient single-page recovery of all pages in a B-tree index, non-leaf pages, typically only 1–0.1 % of all pages in a B-tree, should be fully logged such that existing recovery techniques suffice, e.g., log-based single-page recovery [17].

Recovery from single-page failures requires an earlier image of the page plus a continuous list of detailed log records for all changes between this "backup" image and the present. Allocation-only logging is advantageous for traditional index creation and for adaptive merging precisely because it avoids details in log records. Thus, it seems that allocation-only logging is incompatible with efficient, "pin-point" recovery from single-page failures. However, as we describe below, partitioned B-trees and adaptive merging enable single-page recovery without such log records, namely by re-deriving lost contents from retained prior data. This is due to index operations proceeding in distinct simple steps with valid and useful states in between, even if each merge step merges only a small key range.

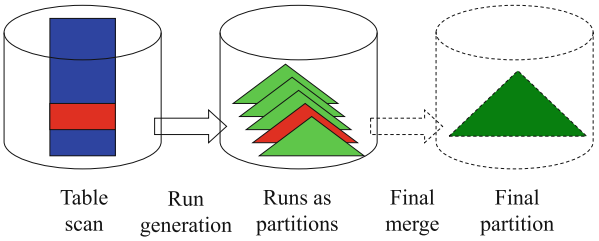


Fig. 10 Single-step recovery: run generation (color figure online)

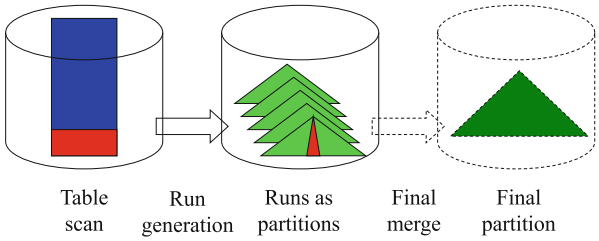


Fig. 11 Single-page recovery: run generation

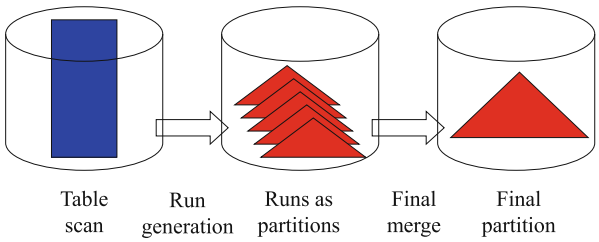


Fig. 12 Single-step recovery: merging

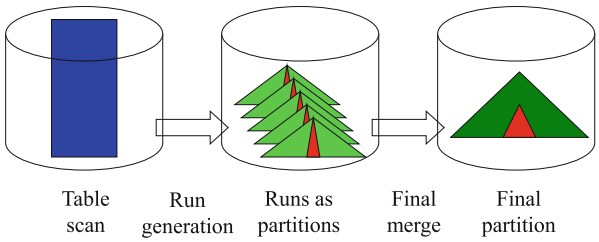


Fig. 13 Single-page recovery: merging

Should one of the initial runs become unreadable, it can be recovered if the appropriate part of the primary data structure can be identified, retrieved, and re-sorted. If only a key range within an initial run becomes unreadable, this key range translates to a predicate when re-scanning the primary data structure, which reduces the sort effort but not the scan effort.

Figure 10 illustrates the technique, where recovery of a single run (center, red) reaches back to the original table (left, blue) but scans only a part of it (left, red). The final index (right) does not participate in this scenario. It might not even exist yet and is thus drawn with dashed lines.

Such recovery works very efficiently if each partition in the new index maps to a specific segment of the source data structure. Ideally, a table's primary data structure is a B-tree index (a clustered index also known as index-organized table), the scan providing input to run generation uses the index order (as opposed to an allocation-order scan), and run generation proceeds in read–sort–write cycles (e.g., using quicksort, not using a continuous process such as replacement selection). In this case, the read–sort–write cycles and the index order scan provide a simple mapping from a run in the new index to a key range in the data source, and the primary index provides efficient access to just that key range. In contrast, run generation by replacement selection permits only less precise mappings, and an allocation-order scan or a primary data structure other than an index requires an unusual predicate on a page range rather than a standard predicate on a key range.

If only a single page within a run is unreadable, it can be re-derived efficiently using a partial scan of the original table. Differently from the partial table scan in Fig. 10, this partial scan applies a predicate matching the key range of the unreadable page. If a B-tree represents each run or if a single partitioned B-tree represents all runs, the parent page in the B-tree structure can provide the required key range.

Figure 11 illustrates recovery of a single page in an adaptive merging partition. Scanning the appropriate fraction of the data source quickly produces the index entries that belong to the unreadable page of the index partitions.

When the number of partitions that produce results exceeds the number that can be merged in a single step, then instead of merging results into the final partition, some results may be merged into new partitions. This is similar to the intermediate runs produced by an external merge sort. In such cases, additional queries may be required before these records are adaptively merged into the final partition, or if they are not queries again, they may remain where they are.

Should an intermediate run or a key range within such an intermediate partition become unreadable, recovery repeats the merge logic for that key range. The same is true for the final merge step producing the final, fully optimized index: Should a part of the final index partition become unreadable, it can be recovered by re-merging data from the final runs.

For example, Fig. 12 illustrates single-step recovery from intermediate run partitions, i.e., it complements the single-step recovery illustrated in Fig. 10. If intermediate runs still exist, recovery of the final index can omit table scan and run generation, instead repeating only the final merge step.

Figure 13 illustrates single-page recovery by partially repeating a merge step. If a single page (or a small set of pages) is unreadable in the final index partition, intermediate runs stored in a B-tree permit direct access to the required key range. A short merge operation can reproduce precisely the unreadable key range without wasting any effort on other key ranges.

During merge steps in a partitioned B-tree, e.g., during adaptive merging, a limited merge fan-in reduces the memory

allocation required for the side effect of query execution. Thus, there is a trade-off between efficiency of a merge step (favoring a high merge fan-in) and the overhead of memory allocations (favoring a small merge fan-in).

In summary, efficient single-page recovery for index operations does not require logging the new index contents. Instead, it merely requires retaining data structures, i.e., delaying their removal from temporary storage space. Doing so enables efficient recovery of both large and small failures, e.g., single-page failures in intermediate data structures (e.g., runs during index creation) and in final index structures. While the prior design for single-page recovery requires extensive logging, the new design relies on data structures created in the standard sequence of steps. Online index operations, i.e., those database updates during index creation, require additional application of single-page recovery techniques, as detailed elsewhere [17].

5 Adaptive indexing for column-oriented databases

In this section, we study the implications of concurrency control for adaptive indexing in a column-store environment. Adaptive indexing was originally proposed as a column-store-specific index mechanism in the form of database cracking [26] and has subsequently evolved to further column-specific refinements such as sideways cracking [28] and hybrid adaptive indexing techniques [29]. Given that the same core principles apply for all adaptive indexing methods, for simplicity of presentation, our discussion focuses mainly on selection cracking [26].

5.1 Column-oriented storage and access

The storage and access patterns significantly affect the way concurrency conflicts appear and how they can be resolved. In a column-store system, data are stored one column at a time; every attribute of a table is stored separately as a dense array. This representation is the same both in memory and on disk. All columns of the same table are aligned, which allows for efficient positional access to collect all values of a given tuple. For example, all attribute values of tuple i of table R appear in the "i-th" position in their respective column. Such an example is shown in the left part of Fig. 14.

During query processing, the system accesses one column at a time in a bulk processing mode. The right part of Fig. 14 shows the steps of evaluating a simple select project query in a column-store system. It first evaluates the complete selection over one column. Then, given a set of qualifying IDs (positions), it fetches only the required values from another column before computing the complete aggregation in one go again.

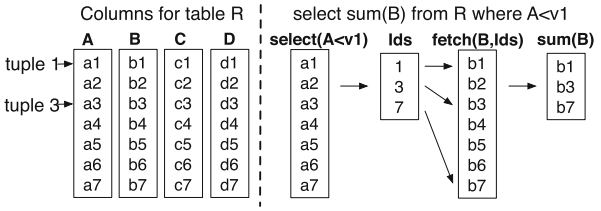


Fig. 14 Storage and access in a column-store system

There are two column-store-specific features that adaptive indexing exploits. First, given the underlying representation of data in the form of fixed-length dense vectors, index refinement actions can be implemented very efficiently. Second, due to bulk processing, each column referenced in a query plan is actually used for only a brief period of time compared to the total time needed to process the complete query. For example, as the right part of Fig. 14 shows, column A is relevant only for the select operator and is not used for the remainder of the query plan. This means that adaptive indexing only needs to use short-term latches that do not necessarily span the whole duration of a query plan.

5.2 Algorithms and data structures

In this subsection, we dive deeper into the details of original database cracking [26] to highlight the design issues and data structures that impact concurrency control.

Database cracking relies on continuous but small index refinement actions. Each such action reflects a data reorganization action of the dense array representing the cracking index. In its original design, the cracker index for a column consists of two data structures: (1) a densely populated array of rowID–value pairs that holds an auxiliary copy of the original column of key values, and (2) a memory resident AVL tree that serves as a table of contents to keep track of the key ranges that have been requested so far. Each select operator call uses the AVL tree to identify the parts of the index that need to be refined. Figure 15 shows the basic cracking array and AVL tree in action as it is affected by two consecutive queries. The first query finds a vanilla uncracked col-

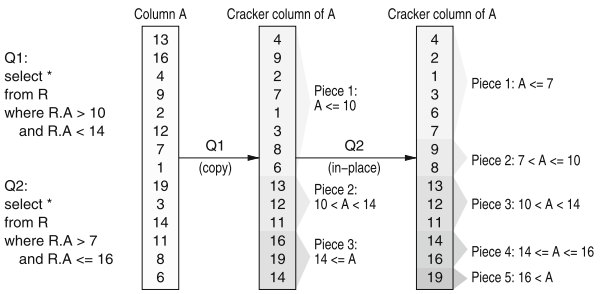


Fig. 15 Database cracking data structures

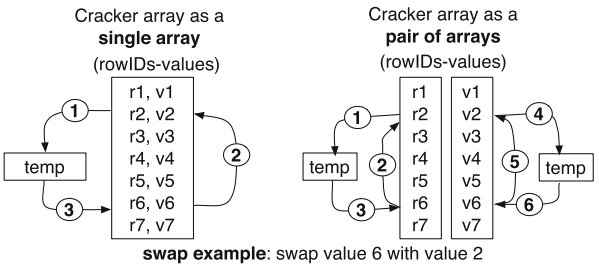


Fig. 16 Cracker array implementation and swap actions

umn and cracks into two new pieces, while the second query may exploit and enhance the existing cracking information by introducing more pieces.

The array is continuously physically re-organized (incrementally sorted based on key values) as a side effect of query processing. The nodes in the AVL tree point to the segments ("pieces") in the cracker array where requested key ranges can be found after the respective reorganization step. Thus, the AVL tree provides instant access to previously requested key ranges and restricts data access as much as possible for the case of a non-exact match, pointing to the shortest possible qualifying range for further cracking.

The latest generation of the cracking release uses a different format for the cracker array. Instead of using an array of rowID–value pairs, it uses a pair of arrays. In the latter case, we have the *rowIDs array* and the *values array*. Figure 16 shows an example comparing both representations. Maintaining separate areas can improve query processing performance, e.g., by providing better cache locality for operators that need to access only the rowID array or only the value array.

5.3 Concurrency control

It is sufficient to use rather short-term latches on the cracker array, the AVL tree, and some global data structure that keeps track of which cracker indexes do exist.

5.3.1 Column latches

For example, consider simple queries that only perform a single selection over a single column; such a query consists of a single select operator that in a bulk mode consumes the entire column and produces the result. When the select operator starts, it first latches the global data structure to check whether a cracker index has already been initialized for the given column. If not, it initializes the respective raw cracker index for that column and latches both the AVL tree and the cracker array. If the cracker index already exists, it latches the AVL tree and the cracker array. Once the latches are acquired, the global data structure can be released, and the select, including any cracker array refinement, is performed with exclusive access to the cracker array. As soon as the select operation, including the necessary array refinement and AVL tree updates, finishes, the index latches can be released.

In case of operator-at-a-time bulk processing as in MonetDB, the select must finish before any other operation in the query plan (that uses the selection result) can start. While using simple coarse-grain per-column latching, this approach benefits from the fact that (1) the latches need to be held only while the select operation is active, and (2) as more queries are processed, both the selection itself and the index refinement benefit from the continuously improving index, shortening the length of the critical section.

5.3.2 Read–write latches

A more complex scenario is when the same column used for selection (cracking) by one query is also used for aggregation by another query. Reorganizing an array that is being concurrently processed by an aggregation operation that reads every tuple within a qualifying range (e.g., sum or average) may lead to incorrect aggregation results. However, multiple aggregation operations may proceed in parallel over the same column. For this reason, we distinguish between read and write latches. Every cracking select operator requires a write latch over the relevant column; all other, non-cracking, operators require read latches so as to protect the data being read by concurrent cracking operators.

5.3.3 Example of column latches

The upper two-thirds of Fig. 17 illustrates how column latches work for three example queries that arrive concurrently and request access to the same column. For each technique being illustrated, the figure depicts when each query acquires a read (blue dashed line) or write (red solid line) latch. For example, reading the first example ("column latch," top row) from left to right, the three queries arrive concurrently, each requesting to compute a sum over a target range. Thus each query will first crack and then aggregate over the qualifying range. Initially, all queries request a write latch but only one may proceed, (Q1 in our example). When Q1 finishes with its crack select operator, Q2 wakes up and starts cracking the column for its own value range. Q3 is still asleep waiting for a write latch to also perform cracking while Q1 blocks as well, as it needs a read lock for the aggregation but cannot proceed as Q2 is now cracking the column. When Q2 finishes with its crack select, both Q1 and Q2 acquire read latches and can now perform their aggregation operators in parallel. After this step, Q1 and Q2 are finished and Q3 may take a write latch and subsequently a read latch to perform its cracking and aggregation, respectively.

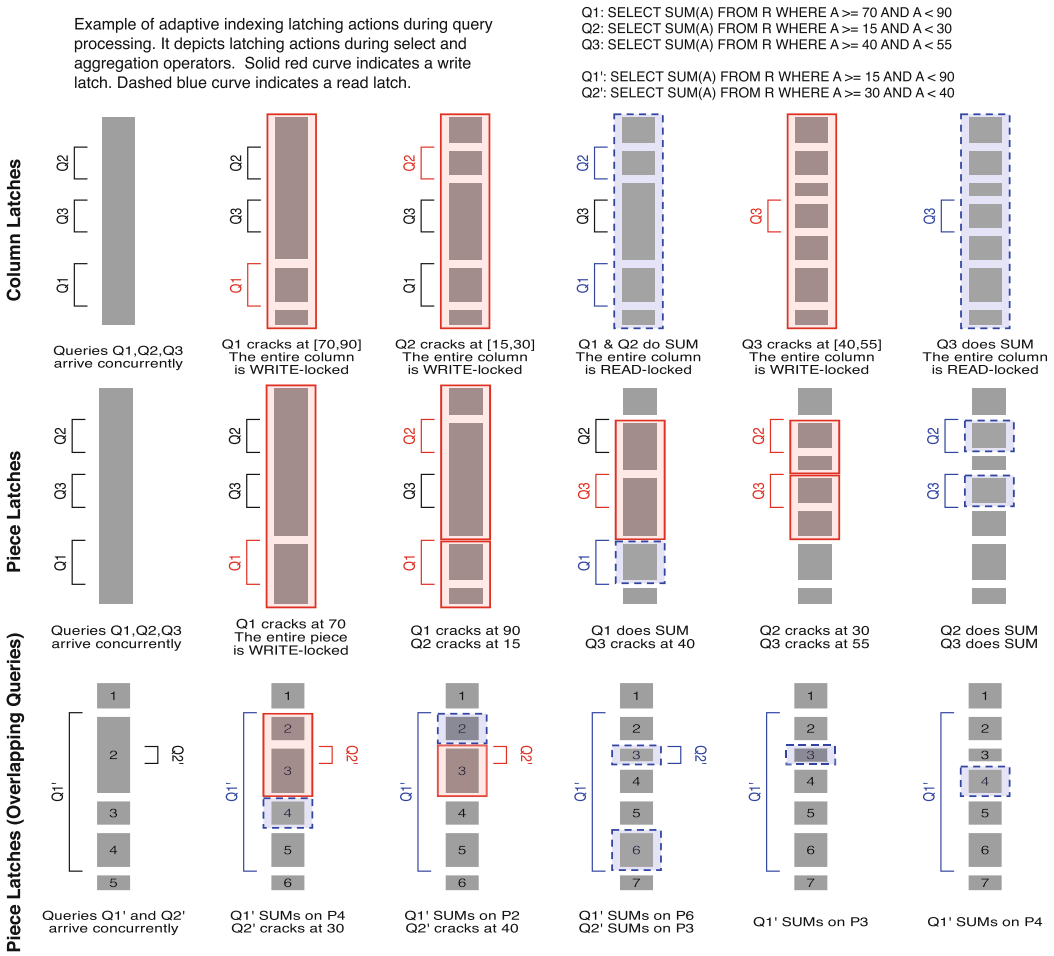


Fig. 17 Concurrent queries with adaptive indexing

5.3.4 Piece-wise latches

As illustrated by the lower two-thirds in Fig. 17, a natural enhancement is given by the fact that the index refinement of adaptive indexing in general and database cracking in particular only accesses a fraction of the index that has not yet been optimized for the requested key range. Hence, only the requested key range needs to be latched both in the cracker array and in the AVL tree. In fact, only the two pieces (segments) that contain the boundary values of the requested key range are physically reorganized. All pieces in between are fully covered by the requested key range, and thus not touched by the cracking select operator.

Figure 18 shows an example where a new query in an already cracked array has to touch only two pieces; only the pieces where its low and high selection bound falls in. This results in a new array, which is now cracked on the low and high bounds as well.

Hence, only the re-organization of the two boundary pieces needs to be protected by exclusive read–write latches, increasing the potential of concurrency even more. With

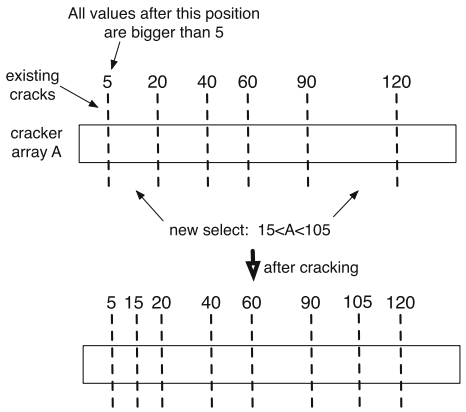


Fig. 18 Only need to touch two pieces during cracking

piece-specific latches, two or more concurrent queries may proceed to crack the same column concurrently as long as they are cracking different pieces of the same column. Similarly, two or more queries may proceed to crack and perform aggregation on the same column concurrently, so long as they operate on different pieces; each distinct column

piece can be accessed by one query at a time for cracking, while it can be accessed by multiple queries concurrently for aggregation.

5.3.5 Optimizations

An additional optimization is that the two cracks needed for each range select may be performed concurrently if they are independent. For example, in Fig. 18, the cracking action for bounds 15 and 105 can happen in parallel as they operate on different pieces. This way, even if there is a conflict for one of them, the query actually proceeds with the second bound.

A crucial detail is that when two or more queries wait for a write latch over the same cracking piece, then upon waking up, the next query needs to re-determine its own bounds as the underlying piece has changed because of the previous query. The illustration in Fig. 19 shows the various cases that may occur. Three queries need the same piece but only one can proceed. Once Q1 has finished, the structure of the underlying piece has changed, and Q2 and Q3 must reevaluate which area of the array they need to crack and where they need to latch. Every query achieves that by walking through the pieces of the array (the leaf nodes of the AVL tree) starting from the original piece they tried to latch. For each piece, they check whether their bound is in this range and if yes, they try to latch this piece; otherwise, they go on to the next. In Fig. 19, Q2 still falls inside the original piece while Q3 is on the next one. In addition, given the creation of new pieces, now Q2 and Q3 may run in parallel as they no longer conflict.

Another optimization has to do with scheduling waiting queries in order to increase the concurrency potential. For example, assume a piece with bounds on [0–100] and 5 waiting queries that want to crack on bounds Q1:20, Q2:30, Q3:50, Q4:70, Q5:90. The worst-case scenario is if they wake up in the order of their requested bounds; e.g., Q1 wakes up first, then Q2, then Q3, etc. This scenario has the lowest potential for concurrency because the remaining queries must always wait. However, if Q3 runs first, the domain is split in half and the remaining queries may run in parallel. Our implementation uses a queue for each waiting query list in a given piece and will insert in the queue the queries with an insertion sort on their bounds. Then once the currently running query finishes, the next one will be the one which is in the middle of the queue.

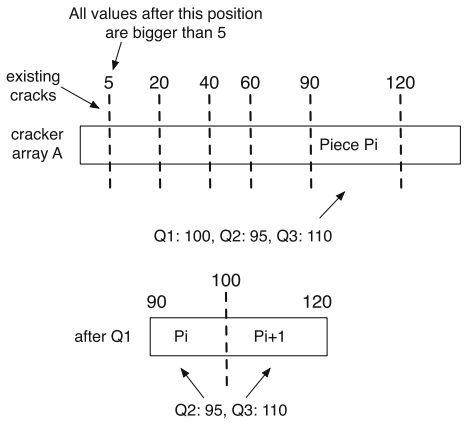


Fig. 19 Increasing concurrency with piece latching

Algorithm 2 CrackWithPieceLatches(c,v)
Crack column c on value v using piece latches and such that all tuples with values lower than v are in a contiguous area on c.

```
1: readLatch(Tree);
2: piece= Tree.findPieceContaining(v);
3: unlock(Tree);

4: //Once we get the piece latch, check whether we got the correct piece;
   might have cracked the same piece in the meantime.
5: while true do
6:    writeLatch(piece);
7:    if piece.contains(v) then
8:       break;
9:    while piece.doesNotContain(v) do
10:      unlock(piece);
11:      readLatch(Tree);
12:      piece = Tree.getNext(piece);
13:      unlock(Tree);

14: //If the piece has not been cracked on v before, then crack it and
   update the index Tree.
15: if piece.hasNotBeenCrackedBeforeOn(v) then
16:    position = CrackColumn(c,piece,v);
17:    unlock(piece);
18:    writeLatch(Tree);
19:    addIndex(Tree,v, position);
20:    unlock(Tree);
21: else
22:    unlock(piece);
```

5.3.6 Algorithms

Algorithms 2 and 3 show the process of exploiting piece latches in more detail for a crack select operator and for an aggregation operator, respectively. For the select operator in Algorithm 2, the main effort is in securing that we get the latch on the proper piece, and then once we crack the piece, we need to also update the tree which maintains all the metadata. For the tree, we use a separate latch such as to allow various operators to search the tree in parallel. Notice also how we need to repeatedly check that we got the correct piece. In more detail, we initially search the tree for the piece which contains the pivot on which we want to crack (i.e., the bound of a select operator). This happens in lines 1–3 in Algorithm 2. However, we still need to verify that this is indeed the right piece. If another concurrent query/operator cracked this piece in the meantime (while we searched the tree and tried to lock the piece), then we might have to move to one of

Algorithm 3 SumWithPieceLatches(c,vlow,vhigh)
Perform a sum aggregation on a crack column c in the range $[vlow, vhigh]$ using piece latches.

```
1: readLatch(Tree);
2: pieceLow = Tree.findPieceContaining(vlow);
3: unlock(Tree);

4: //Once we get the piece latch, check whether we got the correct piece;
   might have cracked the same piece in the meantime.
5: while true do
6:    readLatch(piece);
7:    if piece.contains(vlow) then
8:       break;
9:    while piece.doesNotContain(vlow) do
10:      unlock(piece);
11:      readLatch(Tree);
12:      piece = Tree.getNext(piece);
13:      unlock(Tree);

14: //Go through all pieces and update the aggregation result for each
   piece until we reach the piece that contains vhigh.
15: sumResult = 0;
16: while piece.doesNotContain(vhigh) do
17:    sumResult = sum(c,piece,sumResult);
18:    next = Tree.getNext(piece);
19:    unlock(piece);
20:    piece = next;
21:    readLatch(piece);
22: sumResult = sum(c,piece,sumResult);
23: unlock(piece);

24: return sumResult;
```

the adjacent pieces, i.e., because the current piece has been cracked in possibly multiple new pieces and it may happen that the value we are looking for now resides to one of the new pieces. This happens in lines 5–13 in Algorithm 2 where we repeatedly may move to adjacent (next) pieces until we manage to lock the piece which contains the desired bound. The getNext() method returns the piece, which is immediately adjacent to the current one in the column and which contains values higher than the current piece. Then, we simply crack this piece (line 16), update the tree (line 19), and unlock everything. If a past query has already cracked on the same pivot, then we do nothing of the above (line 22).

Similarly, for the aggregation operator, we need to acquire read latches for all relevant pieces in the needed value range. Once we get the first piece correctly, we can simply go piece by piece until we find the last one; even if a piece is cracked in between, it is safe to get the next piece in the range as we are going to go through all existing pieces in the range anyway.

In both operators, we maintain a read or write latch for at most on piece at a time, allowing other concurrent operators to work in parallel on the rest of the pieces of this column, even if they need to work on overlapping value ranges from the scope of the full query.

5.3.7 Example of piece-wise latching

The middle third of Fig. 17 illustrates piece-wise latching using exactly the same queries as the top part of this figure, which illustrates column latching. As before, Q1 initializes and latches the entire, as-yet-uncracked, column. However, once Q1 has completed the cracking for its low bound, Q2 may proceed to start cracking for its own low bound, while Q1 is cracking for its high bound concurrently. This is possible as after the first crack on the low bound of Q1, two independent pieces have been created. Subsequently, while Q1 is computing its aggregation with a read latch on its qualifying piece, the rest of the queries keep cracking the other pieces of the column.

The bottom third of Fig. 17 depicts one more example of piece latching, where the requested ranges may vary across the incoming queries. With piece latching, cracking and aggregation queries may work concurrently so long as each cracking query has exclusive access to the piece being cracked. Two queries may crack different pieces concurrently, and two queries may perform aggregations in parallel in the same piece.

5.3.8 Continuously reduced conflicts

As the piece-wise discussion indicates, e.g., Fig. 18, the pieces on the cracker array become smaller as the workload progresses. This is the very reason why adaptive indexing enjoys improved performance as we process more and more queries upon a given column. As the pieces of the index become smaller, we achieve both better filtering and also finer granularity of latching. Together, these factors make the task of refining the index increasingly less expensive. Regarding concurrency conflicts, this means that the period of time for which a query needs to hold the write latches decreases over time, which in turn allows more queries to run in parallel. In this way, the concurrency potential improves in an adaptive way; the more important a column is for the workload, the more chances appear to exploit concurrency as the workload evolves.

5.3.9 Other adaptive indexing methods

The techniques presented here apply as is to the rest of the column-store designs for adaptive indexing which we introduced in [29]. This is because the ideas in [29] maintain the same underlying philosophy and follow the same column-store model. In addition, in future work, we discuss interesting opportunities on how the status of the system during concurrency control may trigger new algorithm designs to improve performance even more, mainly by allowing for dynamic strategies which are driven by concurrency conflicts.

5.4 Recovery

The crucial observation is that adaptive indexing (for read-only queries) performs only structural modifications of auxiliary data structures, but leaves the primary data content unchanged. This way, we only need to focus on the auxiliary cracker data structures in respect with transaction control.

The basic refinement step of database cracking is the act of swapping two values in an unordered key range of a dense in-memory cracker array. This step must be an atomic operation. It either happens completely or not. Otherwise, we might leave the index with an incorrect set of values. For this reason, we need to revisit the design choice of how the cracker array is physically represented, i.e., as an array of rowID-value pairs or as a pair of a rowIDs array and a values array. Figure 16 shows an example of how value swapping is done in both representations. In the first case, it is much easier to guarantee atomic data swapping at the software level given that we operate on a single data structure. In the second case, with a pair of arrays, we need to rely on hardware support, i.e., transactional memory to achieve atomic swapping [23]. In both cases, this ensures that such basic data swapping do not compromise the (structural) consistency of a cracker index.

Consequently, if a query is aborted while executing the select operator, hence, the array reorganization is active, but before the AVL tree is updated to reflect the completed refinement for the requested key range, no recovery actions—neither undo nor redo—are required. Some key values might have changed their position, but only within a yet unordered key range. Hence, there is no need to either undo these change or finish the not yet registered sub-partitioning of a key range.

Likewise, in case a query needs to be aborted after the selection operator and index refinement has finished, and the AVL tree has been updated, no recovery is required either. In this case, the only impact of the aborted query is a (slightly) more optimized index structure. This is not harmful, but rather an advantage for future queries.

5.5 Logging

For similar reasons as discussed above, the log volume can be kept at a minimum. In particular, there is no need to log each single swap operation during the index refinement. Instead, if desired, it is sufficient to log the requested key ranges. In case a cracker index is lost in a system crash, the requested key ranges provide all the necessary information to rebuild the cracker index. However, since the index contains only auxiliary data to improve query performance, there is no need to exhaustively log all requested key ranges or to ensure any particular order in the log. For adaptive indexing, logging and crash recovery of the adaptive index are opportunistic. Given the rather low overhead and efficient adaptation of adaptive indexing, we can even afford to not log at all, but rather abandon adaptive indexes after a crash and re-start their adaptive construction as side effect of normal query processing after the system restart.

In some cases it is necessary to maintain a complete log of all cracking actions in a particular column. For example, when we expect to crack many columns of a multi-column table then it is crucial to maintain the columns physically aligned. That is, during query processing, all relevant tuples of more than one columns used in the same query plan should be in the same positions across all columns used. This problem is described in [28]. One solution would be that we always crack and reorganize all columns of a given table. This is not useful though as it would mean that every query would have to touch all columns even if only a few of those are needed at a time. By maintaining a complete log we are able to adaptively apply the cracking actions and physically align all needed columns just before they are going to be used by a new query and thus minimizing the costs of alignment and amortizing the overhead across many queries. Without a log, we would either have to replay all past cracking actions of all past queries before each tuple reconstruction action in every future query or we would have to resort to expensive join actions to reconstruct tuples. Both of these options impose a significant overhead while simple log as described in [28] allows for just-in-time self-organizing tuple reconstruction.

6 Experimental analysis

In this section, we report on a first implementation of concurrency control and recovery in adaptive indexing. The space of research is very broad, as we described in the previous sections. Here, we concentrate on the case of testing a column-store implementation of adaptive indexing using a full existing implementation of database cracking in the MonetDB open-source column-store.

Set-up. The setup in the following experiments is as follows. We use a table of 100 million tuples populated with unique randomly distributed integers. The crucial part of adaptive indexing concerning concurrency is the index refinement as side effect of the selection over a base table. Consequently, to focus on this, we use simple range queries of the form.

Q1: select count(*) from R where $v_1 < A_1 < v_2$
Q2: select sum(A) from R where $v_1 < A_1 < v_2$

The important difference between the two query types is that the second one has to do more work, i.e., both aggregation and selection/cracking.

In order to gauge the impact of concurrency on performance, we increase the number of clients submitting queries concurrently. We use lightweight queries in order to make

the overhead of concurrency more prominent. The effect of more complex queries on adaptive indexing, e.g., TPC-H, can be seen in [28].

We use a 3.4 GHz Intel Core i7-2600 quad-core CPU equipped with 32 KB L1 cache and 256 KB L2 cache per core, 8 MB shared L3 cache and 16 GB RAM. The operating system is Fedora 14.

6.1 Basic performance

This first experiment establishes context by illustrating the basic trade-offs of adaptive indexing as distinct from any concurrency related overhead. It demonstrates its benefits and the scenarios where it can be useful. In this experiment, all queries are ran sequentially and there are no concurrency control mechanism which are active and thus there is no concurrency control overhead for any of the tested approaches.

The scenario is a completely dynamic environment. We assume no workload knowledge and idle time to prepare the system. The only given is that the data are assumed loaded in its basic form. Immediately after the data are loaded, queries begin to arrive in a steady stream with no "think-time."

The experiment compares three approaches using queries of type Q1. In the default case, the system accesses the data using plain scans, with no indexing mechanism present. At the other extreme, we consider the case of a very active approach that resembles a traditional indexing mechanism: When the first query arrives, we build the complete index before we evaluate the query, which can then exploit this index. The benefit is then available to all future relevant queries. In our implementation over a column-store, it is sufficient to completely sort the relevant column(s) and then use binary search to access them.

We use adaptive indexing via a complete implementation of database cracking over MonetDB. Query processing operators reorganize relevant columns and tree structures on the fly to reflect the knowledge gained by each query. All changes happen automatically as part of query processing and not as an afterthought.

Figure 20a compares the basic performance of these three approaches in terms of per-query response time for running 10 queries serially one after the other through a single database client. The queries use random range predicates with a stable 10 % selectivity. The default scan-based approach has a rather stable behavior. The first query is slightly slower, fetching the data from disk. The full indexing approach, labeled "sort" in the figure, shows a significant overhead when building the index with the first query and then enjoys great performance from the second query onwards.

The problem with the scan approach is that it does not exploit past knowledge, resulting in relatively slow performance throughout the span of a workload. The problem with the full indexing approach is that it significantly penalizes

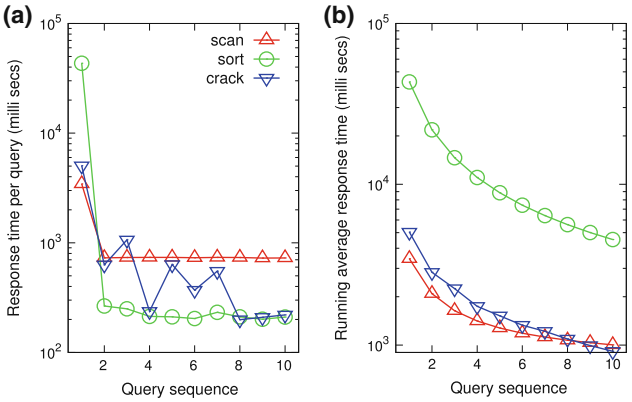


Fig. 20 Basic performance for sequential execution

the very first query. If this query were an outlier, or if the workload span turned out to consist of only a few queries, then this extra overhead may never pay-off. Figure 20b visualizes this by depicting the running average response time for the same experiment. Ten queries are far from enough to amortize the high investment of building the full index with (or before) the first query.

Adaptive indexing solves the above problems in dynamic environments. Figure 20a shows how it maintains a lightweight first touch to the workload, but at the same time, it continuously learns and improves performance in a seamless way, without over-penalizing queries. Performance improves continually and almost immediately in response to the workload. The more queries arrive, the more performance improves. Figure 20b confirms that the low initial investment pays back quickly; after only 8 queries, the initial investment has paid off and the average per-query response time of adaptive indexing becomes less than that of a basic scan approach.

The performance seen in this experiment is representative of the adaptive indexing behavior. Depending on the type of queries posed, the data, and query distribution, adaptive indexing may converge to optimal performance faster or slower in terms of number of queries required. The interested reader can refer to previous papers for in-depth analyses regarding multiple parameters, e.g., skew, updates and multi-column indexes. [26–28,12,14,16,29]. In the rest of the following analysis, we focus solely on concurrency control issues.

6.2 Concurrency control

Let us now focus on how concurrency control impacts performance. For ease of presentation, in this section, we first present a broad analysis. For adaptive indexing, concurrency control is achieved by using the piece latches approach. Then, the next section dives deeper into analyzing the behavior for various parameters and it also presents piece latches in more

detail as well as it benchmarks piece latches against column latches..

The setup of this experiment is as follows. We run a sequence of 1,024 random range queries of 0.01 % selectivity and of type Q2. We perform several runs of the same 1,024 queries, and each time we increase the number of concurrent streams. Each run is completely independent and does not affect previous or future runs; it starts with a clean environment. In more detail, the setup is as follows. We test the serial case where one client runs all 1,024 queries, one after the other. In addition, we test the case where we use 2 clients that start at the same time and each one fires 512 queries. Similarly, we repeat the experiment by starting 4 clients at the same time and each one fires 256 queries, and so on. We go up to the limit of 32 clients, which is the threshold that our experimentation platform, MonetDB, puts in order to throttle the incoming clients and control the amount of concurrent query threads. For every run, we use exactly the same queries and in the same order. Selectivity is purposely kept high to more clearly isolate the costs of the select operator, i.e., do not let aggregation operators hide any overheads. The next section studies this parameter in more depth.

Figure 21a,b depict the results for plain scan, full indexing (sort) and for database cracking, using piece latches. In both figures, the x-axis lists the increasing number of concurrent clients. In Fig. 21a, the y-axis represents the total elapsed time needed to run all 1,024 queries. In Fig. 21b, the y-axis shows the "inverted" results for the same experiment, i.e., depicting throughput in terms of queries per second rather than total execution time for all 1,024 queries.

For all approaches, we see a rather similar trend, i.e., performance shows a linear decrease in total execution time and consequently a linear increase in throughput when going from one over two to four clients, i.e., up to the number of CPU cores in our system. Then, performance peaks at 8 clients, before leveling out and remaining quite stable up to the case of 32 clients running 32 queries each. We do not perform any special methods here for assigning threads to cores. Each incoming query is assigned a single thread and we let the operating system perform all scheduling actions.

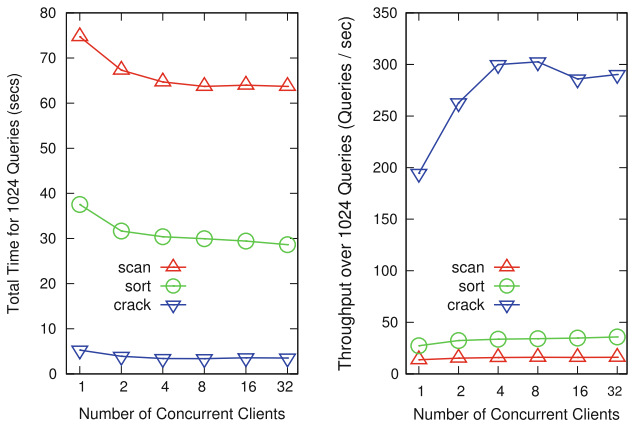


Fig. 21 Effect of concurrency control on total time

The relative behavior between the three different approaches remains the same, regardless the number of concurrent queries. Scan suffers from having to scan the complete column with each query. Full indexing improves over plain scans, but suffers from having to build the complete index via sorting the column. On the other hand, adaptive indexing maintains its competitive advantage and adaptive behavior even with concurrent queries.

We point out that due to their purely read-only data access, neither scans nor binary search actions used in full indexing require any concurrency control during the actual query processing. Adaptive indexing on the other hand has to incur concurrency control costs as it turns read queries into write queries. Nevertheless, its performance remains unaffected.

All in all, the results in Fig. 21 confirm that although adaptive indexing introduces write access for conceptually read-only queries, concurrency is not only possible but also beneficial. Instead of having issues with multiple queries touching the same data, adaptive indexing manages to parallelize queries and benefit from that. The amount of index refinement—and hence the length of the critical part of the query—becomes less and less with every query, quickly vanishing behind the non-critical parts that can be executed in parallel. The next section discusses these issues in more detail.

To get more insight into the overhead of concurrency control and by using the same setup as before, we run the same 1,024 queries using a single client. This way all queries run sequentially, and no concurrency control is required to ensure correct execution. We repeat the experiment twice. For the first run, the concurrency control mechanisms are enabled (piece latches), but for the second run, we disable all concurrency control activities. Thus, the difference in execution time between the two runs is the administrative overhead required for the concurrency control mechanisms of adaptive indexing. What we find is that the concurency control mechanisms add less than 1 % in terms of the total cost to run all 1,024 queries, verifying that adaptive indexing needs only very lightweight concurrency control.

6.3 Detailed analysis

Having seen a generic analysis in the previous section, we now go into more detail to explain the behavior seen under various parameters. Figure 22 depicts the results for our next experiment. We use the same setup as before, i.e., 1,024 random queries and a varying number of clients ranging from 1 (sequential execution) to 32 clients. Here, we also present the behavior of piece vs. column latches as well as we study both queries of type Q1 and of type Q2. In addition, we run

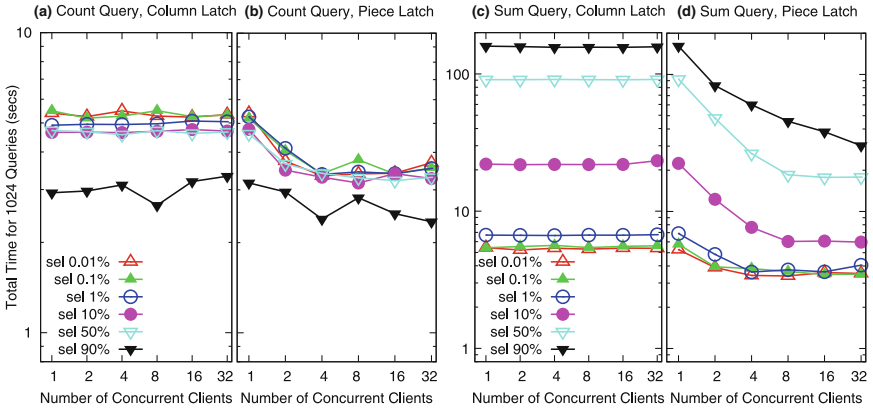


Fig. 22 Column and Piece Latches with count and sum aggregation queries

the experiment for various selectivity factors for each case; queries remain random but selectivity varies. The graphs in Fig. 22 depict the total time needed to run all queries in each case, i.e., the time reported is the time perceived by the last client to receive all answers for all its queries.

Since all queries in this experiment touch the same column, this represents (in terms of the whole workload) the most extreme scenario when it comes to concurrency control as all focus is on a single column, allowing us to stress test adaptive indexing in terms of the maximum concurrency control overheads expected.

Figure 22a,b demonstrates the performance for queries of type Q1 with column and piece latches, respectively. Excluding the low selectivity case (90 %), performance is rather similar for all selectivity runs. This is true both for column and for piece latches. With selectivity 90 % all queries use low and high bounds in their range selection predicates that are focused on only 10 % of the column. As a result, adaptive indexing improves even faster by refining these areas of the column faster compared to other selectivity cases.

When comparing column and piece latches in Figs. 22a,b, we see that piece latches bring significantly more improvements to the adaptive indexing performance. With column latches, performance is rather stable which means that adaptive indexing is not affected by concurrent queries, but at the same time, it does not manage to exploit opportunities for parallelism. This effect is even more noticeable in Fig. 22c,d where we study queries of type Q2. For such queries, an aggregation on the selection column needs to be performed. For this reason, an aggregation operator needs to hold a read latch while going through all qualifying tuples, computing the aggregation. During this time, no cracking can happen and thus no other select operator may run. Only read latches are allowed, e.g., for other aggregation operators of other queries. In the case of column latches, this results in a significant penalty; the whole column needs to be latched.

The lower the selectivity, the higher this penalty as the time needed to perform the aggregation increases (due to more tuples qualifying the selections) and dominates the total

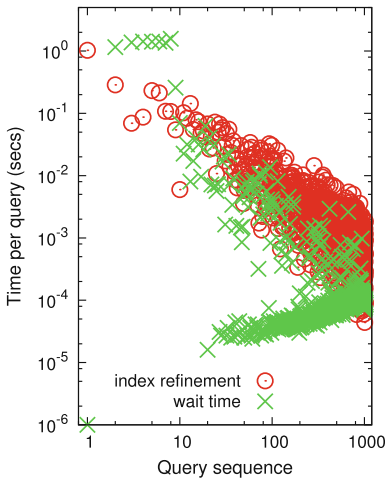


Fig. 23 Breakdown costs

query cost. On the other hand, with piece latches, we allow many queries to run in parallel multiple kinds of previously conflicting operations over the same column as long as they operate on different pieces. Now two queries may crack in parallel two or more different pieces or may crack in one piece and run aggregation on others. This increased parallelism allows piece latching to materialize an even more significant benefit which in the case of Q2 type queries becomes more evident due to the need to maintain read latches for a longer period of time. In this way, this phenomenon becomes more apparent as the selectivity decreases in Fig. 22d.

Figure 23 gives more insight into the above results by breaking down the time of individual queries. It depicts the wait time and the crack time for each individual query as the workload sequence evolves. This is for the case of queries of type Q2 using piece latches with 50 % selectivity and with 8 clients. The wait time is defined as the time each query spends in waiting to acquire a latch. For each query, the number plotted reflects all waiting time, i.e., both for write latches during the crack select operator and for the waiting time for read latches during the aggregation operator. In addition, the time reflects the time needed to acquire all latches for all

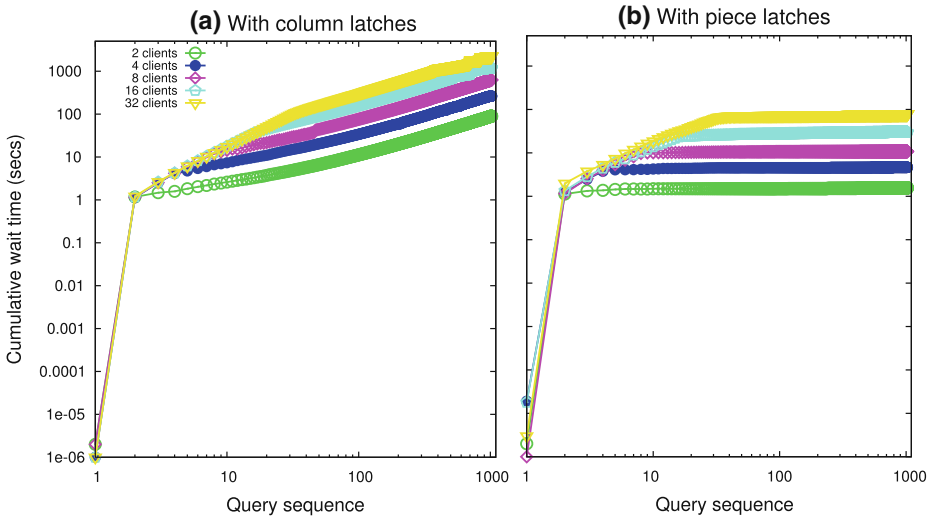


Fig. 24 Wait time evolution through a query sequence

relevant pieces in each operator. The crack time is defined as the time spent purely on refining the index during the select operator (under write latches).

Figure 23 shows that the crack costs follow the behavior, which was observed in past adaptive indexing papers as well, i.e., the more we touch a specific column, the more the index is refined. As pieces become smaller due to more fine-grained indexing, subsequent index refinement operations become faster. This is what brings the adaptive behavior, and Fig. 23 shows that adaptive indexing maintains this behavior even during concurrent queries.

The second observation from Fig. 23 is that the waiting time, i.e., the concurrency control conflicts, shows a similar behavior; it decreases as the workload sequence evolves. Naturally, the very first query does not have to wait at all, depicting a zero cost waiting time in Fig. 23. The next 7 queries though have to wait until the first one finishes cracking the column. This is 7 queries because we use 8 concurrent clients in this experiment and they all have to wait because when the experiment starts there is no cracking index, meaning that the first query has to latch the complete column. Once the first query adds some partitioning, then the concurrency opportunities increase and soon after a few queries have cracked the column, the waiting times decrease.

The main bottleneck in the crack select where the write latches are required is the index refinement time. As this time decreases in Fig. 23, the concurrency conflicts decrease as well. A closer observation on the waiting time in Fig. 23 shows that the wait time almost matches the crack time behavior. For some queries (including the first one), the wait time is minimal as they happen to arrive at a time that the needed piece is free of latches. For the rest of the queries, the wait time follows a continuously decreasing trend similar to crack time; the crack time of one query is in practice the wait time for another query, waiting for a given column piece.

Thus, by using short latching periods and quickly releasing latches as soon as possible, adaptive indexing manages to exploit concurrent queries as opposed to suffering from them. In addition, it is interesting to notice that since adaptive indexing gains continuously more and more knowledge about the data, these latching periods become ever shorter which improves performance even more.

Finally, Fig. 24 depicts the per-query wait time (i.e., the time needed to acquire all required latches) as a function of the total active clients for queries of type Q2 with 50 % selectivity. With more clients being active, the wait time increases. The reason is that with more active clients, more queries arrive concurrently and thus more queries need to block and wait when requesting for a latch. However, as we have seen in previous graphs, at the same time, concurrent queries increase parallelism and thus throughput. For the same reason, we observe that the wait time is higher for the first query (of each client) in the case of piece latches compared to column latches; each query with piece latches has to get 2 latches compared to only 1 in the case of column latches. However, after the first few queries, the wait time per query reduces to a rather low cost for piece latches (the cumulative curve flattens), while for the case of column latches, the cumulative cost keeps increasing; with column latches, every query locks the whole column and blocks all waiting queries. With piece latches, however, each query needs to lock only the relevant piece it needs to crack and at most one piece at a time; as we create more pieces due to cracking, more queries may run in parallel reducing significantly the wait times. For example, with 2 clients, the curve already flattens after 1–2 queries; each query in general creates 1 or 2 new pieces (on the selection bounds); thus, after a couple of queries, there are enough pieces for 2 clients to be able to run concurrently. When we have 4 clients, it takes a few more queries to have enough pieces such that every client may run independently,

while with 30 clients, we need about 30 queries (the column is cracked in about 60 pieces at this point). In this way, this leads to a total benefit of one order of magnitude in wait time for piece latches in the duration of all 1,024 queries. The higher the number of clients, the bigger the benefit for piece latches; while piece latches can exploit concurrent queries to increase parallelism, column latches can only block each concurrent query until the required column is free.

6.4 Recovery

Here, we analyze the impact of the very basic recovery techniques discussed in Sect. 5.4 on the performance of adaptive indexing, i.e., database cracking in this case. There is no concurrency control overhead involved in this test as queries run sequentially. We use the same simple range query as before and run a sequence of 1,000 queries requesting randomly chosen key ranges. To enforce recovery situations, we randomly abort 50 % of the queries. As discussed in Sect. 5.4 we do not perform any recovery action. Consequently, the effect of aborting a query comes in two flavors. In case the query is aborted during the selection and index refinement, but before the AVL tree is updated, the query simply has no (known) contribution to optimizing the index. In case the query is aborted after selection, index refinements and AVL tree updates have finished, "even" the aborted query has contributed its share to optimizing the index. In addition to aborting queries like this, we also simulate a system crash after 500 queries, by abandoning the complete cracker index and restaring it from scratch.

Figure 25 depicts the results in terms of per-query performance. The results indicate that the fact that (some of) the aborted query leave no (known) contribution to the index refinement has only marginal impact on the over all performance. The convergence of the index and hence the improvement of per-query performance is only slightly slower than without query aborts. When the index is abandoned entirely due to the simulated system crash after 500 queries, query performance re-starts at the original level, but improves with more queries just as quickly as before. Obviously, an improved version could flush the entire index to disk whenever there is time to spare or regularly, say, every 100 queries. Then the restart after a crash could load the last index snapshot and proceed from there instead of restarting from scratch. The presented results indicate, however, that even without any explicit recovery actions, query aborts and system crashed result at most in a rather acceptable temporary performance degradation.

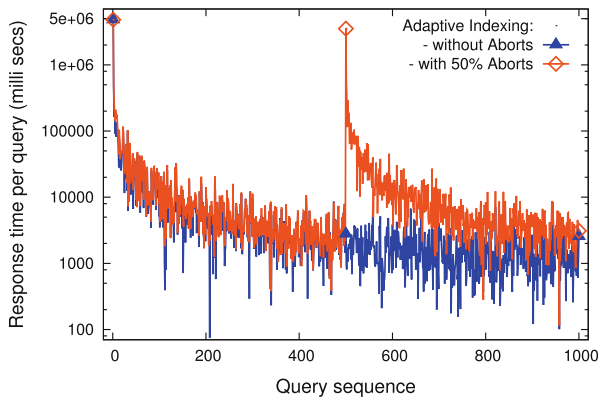


Fig. 25 Recovery: performance per-query (seq. execution)

7 Summary and conclusions

Recent papers have introduced adaptive indexing in the forms of database cracking and adaptive merging. The main idea shared by both techniques is on-demand index construction and optimization as side effects of query execution. At first glance, this seems to turn read-only queries into update transactions, triggering the question whether the anticipated concurrency control and recovery overhead will prohibit the use of adaptive indexing in multi-user scenarios. In this paper, we address this question and show that with judicious application and extension of prior work, concurrency control conflicts and overheads as well as log volume and recovery time can be reduced to practical or even negligible levels.

The key observation is that adaptive indexing applies only *structural* modifications to the physical representation of the index but leaves the logical *contents* of the index unmodified. This relaxes the constraints and requirements during adaptive indexing compared to those considered for traditional index updates. Furthermore, we observe that even those structural changes are optional and propose a new method for partial forward recovery with adaptive early termination during recovery. Using adaptive merging and database cracking as examples, we introduce concrete implementations of our new techniques. The experimental evaluation of our implementation of concurrency control and recovery for database cracking demonstrates that the performance overhead of concurrency control during structural updates is minimal, and that adaptive early termination alleviates problems with both concurrency control and recovery in adaptive indexes.

References

1. Bayer, R., Schkolnick, M.: Concurrency of operations on B-trees. Acta Inf. **9**, 1–21 (1977)
2. Bayer, R., Unterauer, K.: Prefix b-trees. ACM TODS **2**(1), 11–26 (1977)
3. Bruno, N., Chaudhuri, S.: An online approach to physical design tuning. In ICDE, pp. 826–835 (2007)
4. Bruno, N., Chaudhuri, S.: Physical design refinement: The 'merge-reduce' approach. ACM TODS **32**(4), 1–41 (2007)

5. Chaudhuri, S. Narasayya, V.R.: Self-tuning database systems: A decade of progress. In VLDB, pp. 3–14 (2007)
6. Finkelstein, S.J., Schkolnick, M., Tiberio, P.: Physical database design for relational databases. ACM TODS **13**(1), 91–128 (1988)
7. Graefe, G.: Sorting and indexing with partitioned B-trees. In CIDR (2003)
8. Graefe, G.: Hierarchical locking in b-tree indexes. In BTW, pp. 18–42 (2007)
9. Graefe, G.: A survey of B-tree locking techniques. ACM TODS **35**(2), 16:1–16:26 (2010)
10. Graefe, G.: Modern B-tree techniques. Found Trends Databases **3**(4), 203–402 (2011)
11. Graefe, G.: A survey of B-tree logging and recovery techniques. ACM TODS **37**(1), 1:1–1:35 (2012)
12. Graefe, G., Idreos, S., Kuno, H., Manegold, S.: Benchmarking adaptive indexing. In TPCTC, pp. 169–184 (2010)
13. Graefe, G., Kimura, H., Kuno, H.: Foster b-trees. ACM Trans. Database Syst. (TODS) **37**(3), 17 (2012)
14. Graefe, G., Kuno, H.: Adaptive indexing for relational keys. In SMDB, pp. 69–74 (2010)
15. Graefe, G., Kuno, H.: Fast loads and queries. Trans. Large Scale Data Knowl. Cent. Syst. **4**, 31–72 (2010)
16. Graefe, G., Kuno, H.: Self-selecting, self-tuning, incrementally optimized indexes. In EDBT, pp. 371–381 (2010)
17. Graefe, G., Kuno, H.: Definition, detection, and recovery of single-page failures, a fourth class of database failures. PVLDB **5**(7), 646–655 (2012)
18. Graefe, G., Seeger, B.: Logical recovery from single-page failures. In BTW, pp. 113–132 (2013)
19. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann, Los Altos, CA (1993)
20. Halim, F., Idreos, S., Karras, P., Yap, R.H.C.: Stochastic database cracking: Towards robust adaptive indexing in main-memory column-stores. PVLDB **5**(6), 502–513 (2012)
21. Härder, T.: Selecting an optimal set of secondary indices. In ECI, pp. 146–160 (1976)
22. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)
23. Harris, T., Larus, J.R., Rajwar, R.: Transactional Memory, 2nd edition. Synthesis Lectures on Computer Architecture. Morgan &Claypool Publishers (2010)
24. Hoshino, T., Goda, K., Kitsuregawa, M.: Online monitoring and visualisation of database structural deterioration. In IJAC, pp. 297–323 (2010)
25. Idreos, S.: Database cracking: Towards auto-tuning database kernels. CWI, PhD Thesis (2010)
26. Idreos, S., Kersten, M.L., Manegold, S.: Database cracking. In CIDR, pp. 68–78 (2007)
27. Idreos, S., Kersten, M.L., Manegold, S.: Updating a cracked database. In SIGMOD, pp. 413–424 (2007)
28. Idreos, S., Kersten, M.L., Manegold, S.: Self-organizing tuple reconstruction in column stores. In SIGMOD, pp. 297–308 (2009)
29. Idreos, S., Manegold, S., Kuno, H., Graefe, G.: Merging what's cracked, cracking what's merged: Adaptive indexing in main-memory column-stores. PVLDB **4**(9), 585–597 (2011)
30. Kersten, M.L., Manegold, S.: Cracking the database store. In CIDR, pp. 213–224 (2005)
31. Lehman, P.L., Yao, S.B.: Efficient locking for concurrent operations on B-trees. ACM Trans. Database Syst. **6**, 650–670 (1981)
32. Lühring, M., Sattler, K.-U., Schmidt, K., Schallehn, E.: Autonomous management of soft indexes. In SMDB, pp. 450–458 (2007)
33. Mohan, C., Haderle, D.J., Lindsay, B.G., Pirahesh, H., Schwarz, P.M.: ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS **17**(1), 94–162 (1992)
34. Mohan, C., Narang, I.: Algorithms for creating indexes for very large tables without quiescing updates. In SIGMOD Conference, pp. 361–370 (1992)
35. Praveen Seshadri, A.N.S.: Generalized partial indexes. In ICDE, pp. 420–427 (1995)
36. Saracco, C.M., Bontempo, C.J.: Getting a lock on integrity and concurrency. Database Program. Des. (1997)
37. Schnaitter, K., Abiteboul, S., Milo, T., Polyzotis, N.: COLT: Continuous on-line tuning. In SIGMOD, pp. 793–795 (2006)
38. Severance, D.G., Lohman, G.M.: Differential files: Their application to the maintenance of large databases. ACM TODS **1**(3), 256–267 (1976)
39. Srinivasan, V., Carey, M.: Performance of on-line index construction algorithms. In EDBT, pp. 293–309 (1992)
40. Stonebraker, M.: The case for partial indexes. SIGMOD Record **18**(4), 4–11 (1989)
41. Weikum, G.: Principles and realization strategies of multilevel transaction management. ACM Trans. Database Syst. **16**(1), 132–180 (1991)
42. Weikum, G., Schek, H.-J.: Concepts and applications of multilevel transactions and open nested transactions. In Database Transaction Models for Advanced Applications, pp. 515–553. Morgan Kaufmann, Los Altos, CA (1992)