Foundations and Trends® in Databases

More Modern B-Tree Techniques

Suggested Citation: Goetz Graefe (2024), "More Modern B-Tree Techniques", Foundations and Trends® in Databases: Vol. 13, No. 3, pp 169–249. DOI: 10.1561/1900000070.

Goetz Graefe
Google Inc.
GoetzG@google.com

now
the essence of knowledge
Boston — Delft

Contents

More Modern B-Tree Techniques

Goetz Graefe

Google Inc., USA; GoetzG@google.com

ABSTRACT

An earlier survey of modern b-tree techniques is now over a decade old. Obviously, it lacks descriptions of techniques invented and published during this time. Just as importantly, it lacks descriptions of insertion-optimized b-trees in the forms of log-structured merge-trees and stepped-merge forests, which seem to have become almost as ubiquitous as b-trees themselves. This monograph complements the earlier survey in order to bring the combined contents up-to-date.

Goetz Graefe (2024), "More Modern B-Tree Techniques", Foundations and Trends® in Databases: Vol. 13, No. 3, pp 169–249. DOI: 10.1561/1900000070.

1

Introduction

B-tree indexes have been ubiquitous in databases for decades [11]. Their contribution to efficient database query processing can hardly be overstated. An earlier survey of modern b-tree techniques [29] has been widely used for education and reference. Unfortunately, it omits some of the techniques known then and of course those invented since. This monograph is intended to complement this earlier survey and to bring the combined contents more up-to-date.

Section 2 summarizes some of the highlights from this earlier survey [29]. It is intended as a motivation for reading the earlier survey, not as a substitute. Section 3 adds new techniques for tree structures. After a short discussion on in-page formats, a common foundation for further techniques is a reversal from B^{link}-trees and from multiple pointers to each b-tree node. A single pointer to each b-tree node enables write-optimized b-tree, pointer swizzling in a database buffer pool, foster b-trees, self-repairing b-trees, and more. This section also discusses deferred updates and deferred index optimization, i.e., it separates delayed maintenance of index contents and index structure.

Section 4 reviews log-structured merge-forests and stepped-merge forests, topics omitted from the earlier survey [29]. For those, it proposes new storage structures, even b-trees without branch nodes, which probably seems like a contradiction, and new algorithms, including a rather unconventional schedule for merging runs in an external merge sort or for compacting deltas in a log-structured merge-forest.

Section 5 adds a few new techniques in query processing. This includes better use of b-trees on storage and in memory. During index intersection and join, the secondary sort keys enable very efficient merge algorithms without explicit sorting.

Section 6 focuses on concurrency control. The topics include new granularities of locking for fewer invocations of the lock manager and for fewer false conflicts for transactions in serializable transaction isolation – the new technique is called orthogonal key-value locking. They further include shorter enforcement periods of locks, both during execution of the transaction's application logic and during commit processing – the new techniques are called deferred lock enforcement and controlled lock violation. They are not specific to b-trees but the combination of orthogonal key-value locking, deferred lock enforcement, controlled lock violation, and multi-version storage promises to eliminate practically all false conflicts in database concurrency control.

Section 7 covers in-memory b-trees, from thread-private via shared-but-transient to persistent. Among the core considerations are the requirements for concurrency control, logging, and recovery. For in-memory databases and their b-trees, instant reboot combines the techniques of instant restart and instant restore, in effect applying the logic for a double failure in a traditional database with external storage. The final Section 8 sums up and concludes.

2

Modern B-Tree Techniques

The earlier survey [29] starts with basics – the number of leaf and branch nodes in a b-tree, the number of comparisons in a leaf-to-root search (which depends only on the total count of index entries in the leaf nodes, not on the sizes of leaf and branch nodes, except for rounding effects), the value of tolerating some free space to absorb insertions and deletions locally, the efficiency of sorting future b-tree entries before building an index, and more. Importantly, the section on b-tree basics describes the core algorithms for insertion and deletion of index entries and for splitting a node, merging two nodes, and more.

In databases, b-trees are used as primary or clustered indexes and as secondary or non-clustered indexes. The former is a table's essential storage structure with all rows and all columns in the b-tree leaf nodes, whereas the latter is a redundant storage structure with a key-pointer pair in each index entry. The pointer is a record identifier on storage or a search key in the table's primary index; it is a memory address only in the context of in-memory databases. For materialized views, b-trees can be both primary and secondary indexes. For materialized joins, a single b-tree with multiple record types enables both efficient retrieval

of join results and efficient incremental maintenance during insertions, updates, and deletions in one of the join inputs.

A comparison of b-trees and hash tables as database indexes focuses on index creation (sorting vs. incremental insertions) and transactional concurrency control (serializability and phantom protection by locking gaps between existing key values). B-trees on hash values can be competitive with traditional hash indexes by interpolation search within nodes and by caching all branch nodes, ideally pinning those pages in the database buffer pool and swizzling parent-to-child pointers (see Section 7.2). B-trees on hash values also enable efficient query execution, e.g., merge joins on hash values.

Beyond the most basic variants of the b-tree data structure and the most basic algorithms for search and update, the earlier survey reviews a number of techniques not commonly taught, known, or appreciated. The present section attempts to review some of these as an inspiration for the reader who is not familiar with the earlier survey.

The section on data structures and algorithms reviews variable-length records, normalized keys, duplicate key values and their compression up to bitmaps, and other space management topics. It does not cover, for example, b-tree variants optimized for minimal latching and maximal concurrency control among threads (see Section 6).

Figure 2.1, copied from [29], shows six possible representations of three entries in a secondary index. Each entry maps a last name ("Smith") to a row identifier (e.g., 4711). Multiple occurrences of the key value "Smith" can be entirely separate index entries, a list, a list with compression, a bitmap, or even a compressed bitmap. Not included in Figure 2.1 is a compressed bitmap, e.g., by run-length encoding. Also missing is compression of a list of row identifiers by replacing absolute values (as in Figure 2.1(b)) with their numeric differences. Note that these numeric differences basically equal the run-length of "0" runs in a bitmap. While representations, their space requirements, and the complexity of their incremental maintenance may differ widely, transactional concurrency control is the same for all representations.

The section on transactional techniques stresses the difference between logical contents and physical representation of database contents. User transactions read and modify logical contents whereas system

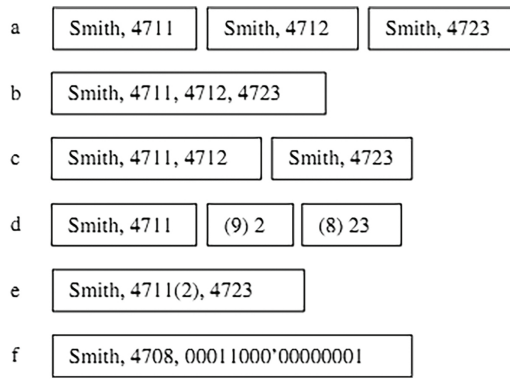

Figure 2.1: Alternative representations of duplicates.

transactions inspect and modify the representation. For example, splitting a b-tree node modifies the database representation including its remaining free space but it does not modify the logical contents as observable with, say, an SQL query like "select count(*) from . . . ." Latches coordinate threads to protect in-memory data structures including in-memory buffers whereas locks coordinate transactions to protect database contents. Among locking techniques for user transactions, the section reviews forms of key-range locking but not orthogonal key-value locking (Section 6.1).

Figure 2.2, copied from [29], summarizes the differences between locks and latches. Lock acquisition and release usually requires ownership of a latch. For example, an execution thread must hold a latch on a database page in the buffer pool while attempting to acquire a lock on a key value within that page; without this latch, another transaction and its execution thread might erase the key value and commit.

The section on query processing stresses the difference between table and index – the former is the description of a logical row type whereas the latter is the description of a physical storage structure. Integrity constraints naturally (and only!) belong to a table definition whereas sort order and compression naturally (and only!) belong to an index definition. In this point-of-view, a "sorted table" and a "unique index" are each self-contradictory. Using this distinction of table and index, many common and uncommon query evaluation plans are discussed,

	Locks	Latches
Separate …	User transactions	Threads
Protect …	Database contents	In-memory data structures
During …	Entire transactions[2]	Critical sections
Modes …	Shared, exclusive, update, intention, escrow, schema, etc.	Read, writes, (perhaps) update
Deadlock …	Detection & resolution	Avoidance
… by …	Analysis of the waits-for graph, timeout, transaction abort, partial rollback, lock de-escalation	Coding discipline, instant-timeout requests, "lock leveling"[3]
Kept in …	Lock manager's hash table	Protected data structure

Figure 2.2: Locks and latches.

e.g., index join to "cover" a query with "index-only retrieval" as well as index-by-index update plans for maintenance of a table with many secondary indexes.

Figure 2.3, copied from [29], shows parts of a query execution plan, specifically those parts relevant to secondary index maintenance in an update statement. Not shown below the spool operation is the query plan that computes the delta to be applied to a table and its indexes. In the left branch, no columns in the index's search key are modified. Thus, it is sufficient to optimize the order in which changes are applied to existing index entries. In the center branch, one or more columns in the search key are modified. Thus, index entries may move within the index or, alternatively, updates are split into deletion and insertion actions. In the right branch, search key columns in a unique index are

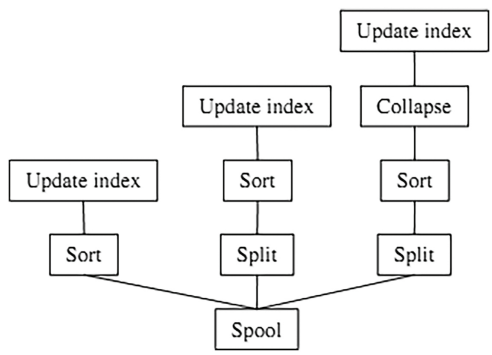

Figure 2.3: Optimized index maintenance plan.

updated. Thus, there can be at most one deletion and one insertion per search key in the index, and matching deletion and insertion items can be collapsed into a single update item, which might save root-to-leaf B-tree traversals as well as log volume. In spite of the differences among the indexes and how they are affected by the update statement, sorting speeds up their maintenance. While sorting for efficient b-tree creation and maintenance are well-known techniques, in-memory b-trees for efficient sorting are perhaps surprising (see Sections 5.1 and 5.2).

The section on b-tree utilities covers not only index creation and index defragmentation but also verification of database indexes, both the structural integrity of a single index and the consistency among related indexes, e.g., primary and secondary indexes of the same table. Consistency checks can be offline with no concurrent database updates, online, or continuous, i.e., integrated into leaf-to-root traversals during ordinary query execution. The latter is particularly useful during maintenance and optimization of the b-tree software. It can be efficient yet comprehensive if the b-tree avoids sibling pointers; instead, each node has fence keys, which are simply copies of branch keys in the parent node.

Figure 2.4, copied from [29], shows a partitioned b-tree with two partitions, the large main partition and a small partition containing selected index entries for efficient deletion in the future. The actual deletion and removal can de-allocate entire leaf pages rather than remove individual records distributed in all leaf pages. If multiple future deletions can be anticipated, e.g., daily purge of out-of-date information, multiple victim partitions can be populated at the same time. Obviously, partitioned b-trees also have a role to play in bulk insertions.

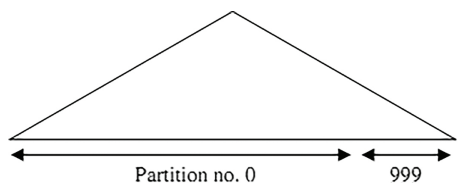

Figure 2.4: Partitioned b-tree prepared for bulk deletion.

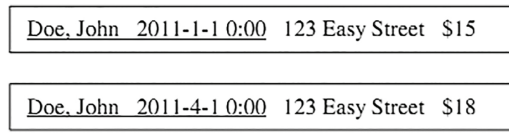

Figure 2.5: Version records with start time as key suffix.

The section on advanced key structures extends the benefits of b-trees and their software development effort to hash indexes (b-trees on hash values), spatial and spatio-temporal indexes (using space-filling curves), master-detail clustering (using merged indexes), column stores (using run-length encoding of row identifiers), and version stores (using key suffixes).

Figure 2.5, copied from [29], shows two version records for the same logical index entry. Each version record carries only its start time, not its end time, which is implied by the start time of the next version. A tombstone record is required when a logical index entry is deleted. A version record is a ghost record if its replacement version is older than the oldest active transaction in snapshot isolation or if it is a tombstone and all prior versions are ghosts.

In summary of the earlier survey [29], b-trees can and should serve as universal storage structure for databases, key-value stores, etc. A b-tree is a sorted array with a cache of key values frequently needed in binary search, plus free space to absorb updates locally; a partitioned b-tree is an external merge sort interrupted and ready to be continued, even one key range at a time.

In summary of Section 2:

- In databases, entries in secondary indexes "point" to rows in primary storage structures by means of on-disk addresses, by search keys in primary indexes, or (within in-memory databases) virtual addresses.
- There are many variations, e.g., b-trees on hash values or on space-filling curves, b-trees with lists of row identifiers or with bitmaps, and more.

- B-trees on uniformly distributed keys such as hash values benefit from interpolation search; b-trees in buffer pools benefit from swizzling pointers to virtual addresses.

- In modern b-trees, high-level concurrency control coordinates user transactions to protect logical contents, whereas low-level concurrency control coordinates threads to protect physical representation and structure. User transactions implement queries and updates of logical contents, whereas system transactions implement structure changes and all space management.

- Required database utilities for b-trees include not only creation and defragmentation but also consistency checking within a b-tree, e.g., key ranges and pointers, and among b-trees, e.g., between a table's primary index and all its secondary indexes. B-tree consistency checking and single-page repair can be embedded in query processing and other b-tree traversals.

- B-trees "cache" the effort spent on sorting index entries during b-tree creation and maintenance, such that queries, e.g., merge join, do not need to repeat the sort effort. A b-tree's root node caches the key values most frequently needed in binary search of the entire collection, i.e., all leaf nodes and their index entries. Free space in leaf nodes is permitted in order to absorb small insertions and deletions efficiently; free space in branch nodes is permitted in order to absorb small structural changes efficiently. (In read-only b-trees, there is no need for free space.)

3

Tree Structure

Many introductions to b-tree techniques dwell on the difference between b-trees and b^+-trees. All implementations of b-trees in databases put all data records in their leaf nodes and use the branch nodes only to guide search and scan operations. The present section covers multiple recent improvements to b-tree structures.

3.1 In-Page Organization and Compression

B-trees and b-tree indexes are sometimes perceived as obsolete – for decision support, statistical analysis, and machine learning, columnar storage often seems more efficient. As a very general rule – with details depending on page size, column count, column sizes, compression, access latency and transfer bandwidth of storage devices, and more – searching secondary indexes loses to column scans if a query requires columns beyond those included in the secondary index or if more than 1‰ of all rows are needed.

In traditional secondary indexes, each index entry comprises a search key and a bookmark. The latter "points" to a record in a primary storage structure – if that is a heap file, the bookmark comprises file identifier, page identifier, and slot number within the page; if the primary storage

structure is a b-tree or other form of index, the bookmark is usually a unique search key there. In non-unique secondary indexes, the bookmark is included in the index sort key. Among other advantages, this permits merging lists of bookmarks efficiently.

In order to speed up query processing using secondary indexes, some database systems support additional column values, sometimes with an "include" keyword in the index definition. Foreign keys are good candidates if speeding up multi-table join queries is important. In those cases, the included columns are appended to each index entry, after the bookmark.

Traditional b-tree pages are designed for simplicity, efficient search, and variable-size index entries. Even fixed-size fields may require variable-size storage if compression is employed. The traditional layout uses an array of pointers (byte offsets within the page), possibly an array of pairs of pointer and key prefix after page-wide prefix truncation [5], [55]. Alternative approaches employ tries [21] rather than binary search or interpolation search [27], [65].

The masstree structure is a unique amalgamation of trie and b-tree: "A Masstree is a trie with fanout 2^{64} where each trie node is a B^+-tree" [57]. It is not directly clear how the masstree compares to a b-tree with prefix- and suffix-truncation [5] after order-preserving compression [3], [73].

With respect to CPU caches, columnar layout within each database page can also have advantages [2]. In b-tree nodes and pages, updates as well as binary search or interpolation search favor the traditional row-by-row format (n-ary storage model – NSM), whereas full-page scans may favor an in-page column format (partitioned across – PAX). A hybrid in-page layout attempts to support both updates and column scans [41]. Its principal ideas are to grow fixed-size and variable-size columns from opposite ends of a page (as in NSM) and to fill each cache line with values from a single fixed-size column (as in PAX).

Figure 3.1, copied from [41], illustrates the hybrid page layout (HPL) for five records with three fields. Most prominent in Figure 3.1 is the single pool of free space between the two allocation spaces for fixed-size values and for variable-size values. This feature is very similar to traditional NSM pages and very different from PAX pages. The

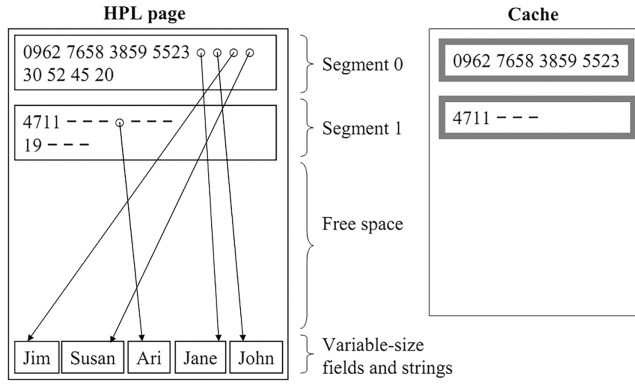

Figure 3.1: The hybrid page layout (HPL).

second obvious aspect is that the variable-size fields are all allocated from the same pool of free space, shown at the bottom of Figure 3.1. This particular example uses only a single variable-size field; if it had multiple variable-size fields, values of different fields would be freely interleaved. In the simplest variant of the hybrid page layout, cache lines are not considered in space allocation for variable-size fields. For fixed-sized fields, shown at the top of Figure 3.1, each segment represents 4 records in this example. Thus, 5 records require 2 segments. The pointers (offsets) for variable-size fields are represented just like user-defined fixed-size fields. Note that the 5 values of each field (e.g., 4-digit employee identifiers) are not contiguous, rather different from the PAX design. However, as in PAX, each cache line contains values from only one field, shown on the right of Figure 3.1, but the cache lines holding the same field are distributed over all the segments within a page.

Columnar storage layout and the focus on cache lines suggest sequential rather than binary search. Efficient sequential search, in particular in b-trees with long or variable-size keys, is possible with fixed-size order-preserving codes. Poor man's normalized keys encode only the first few bytes of each key and are suitable for binary search; for sequential search, offset-value coding [12] captures the first difference between neighboring key values in a fixed-size code. Extending the logic of prefix truncation [5] and of merging, it can exploit modern hardware and its SIMD instructions.

The other central topic for in-page organization is compression in non-unique secondary indexes. Non-unique indexes often pair a distinct key value with a list of row identifiers, which can be compressed using bitmaps or run-length encoding. Very large lists of row identifiers may span multiple b-tree leaves, even after compression. In that case, row identifiers or their leading bytes may become part of a branch key within a b-tree structure. A column beyond b-tree key and row identifiers, also known as an "included" or "stored" column that enables "index-only retrieval," can be compressed using a dictionary or run-length encoding. An extreme design uses zero key columns, i.e., a single list of row identifiers spanning multiple b-tree leaves, plus included column as the only user-visible values in the b-tree. With row identifiers compressed using run-length encoding and the included column compressed by any available method, the resulting b-tree is in effect the core building block of column storage.

In summary of Section 3.1:

- A primary index (or index-organized table) contains entire rows; a secondary index contains some form of pointer to full rows.
- For tables stored on traditional disk drives, search via a secondary index is efficient when selecting less than 1‰ of a table's rows; this might be 1% for tables on SSD drives; and perhaps 10% for tables in non-volatile memory – these rules are very approximate.
- There are many b-tree page layout optimizations for cache efficiency.
- Table and index compression seems a never-ending research topic.

3.2 Learned Indexes

A statistical analysis of key values within a b-tree node can guide the search strategy. For example, if key values are very uniformly distributed, linear interpolation is very effective. At least one relational database product has used this technique to replace binary search with direct access to the correct index entry in some TPC-C tables, notably in indexes on order number and invoice number.

The analysis of key value distributions can be a simple correlation and regression analysis or it can employ machine learning to derive a model of the distribution. The result of such an analysis may divide the set of key values into two or even more subsets such that searching all subsets is faster than searching in the original set of key values. For example, if the set of key values are a sequence but there is a gap within the sequence, the analysis may create two subsets such that choosing among them followed by linear interpolation and direct access within two sets may be faster than binary search over the entire set.

Multiple arguments about learning key value distributions within entire b-tree indexes and within individual pages have been put forth. In addition to search speed, the learned model may require less storage space than the complete set of key values. It seems counter-intuitive, however, that a model can be compressed yet compression of search keys must necessarily less effective. Perhaps the discussion is succinctly summarized in [49], [50] and their references.

In summary of Section 3.2:

- Learned key value distributions can be faster than binary search and interpolation search if searching in the learned synopsis is more efficient than these traditional search methods.
- Compression advantages have been claimed for a learned model versus the set of key values.

3.3 Write-Optimized B-Trees, Foster B-Trees

Indexes including b-trees have been criticized for their high update cost. An insertion of a 10-byte record, for example, might require reading some branch node plus reading and writing a leaf node. Transferring thousands of bytes between memory and storage for a 10-byte record is a tremendous write amplification. Multiple research efforts and designs have attempted to alleviate b-tree update costs.

Write-optimized b-trees [26] address hardware limitations, e.g., block erasure in flash drives and the penalty for small writes in disk arrays [64], whereas foster b-trees [35] improve low-level concurrency control (latching) by temporarily permitting local overflow pages, even in violation of

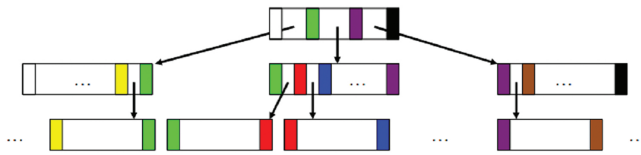

Figure 3.2: A write-optimized b-tree.

logarithmic index depth, and self-repairing b-trees [39] add continuous comprehensive self-checking (if desired) as well as immediate repair using pinpoint access to recovery log and log archive. None of them get to the heart of the write amplification.

Figure 3.2, copied from [35], shows a write-optimized b-tree. Colors indicate key values from $-\infty$ (white) to $+\infty$ (black). The essence of write-optimized b-trees is to avoid side pointers or sibling pointers. With only parent-to-child pointers remaining, moving a b-tree node to a new location requires updating only one database page, namely the parent page. Page movements become inexpensive even without an indirection layer such as a file system or a flash translation layer. Fence keys, i.e., copies of separator keys (or fence keys) in the parent node, permit comprehensive consistency checks, even the correct relationships among cousin nodes, i.e., neighboring nodes that do no share a parent node but only a further ancestor node.

Figure 3.3, also copied from [35], shows the intermediate state during node insertion into a foster b-tree. The left leaf node was split and became a foster parent. In addition to its two fence keys, a foster parent also carries a third key, the foster key, which separates key values in the foster parent and in the foster child. In a foster parent, the high

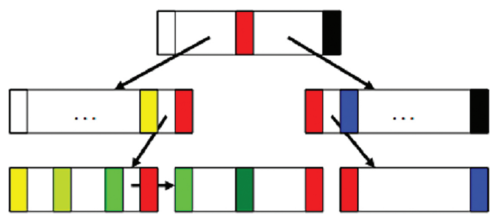

Figure 3.3: Intermediate state in a foster b-tree.

fence key is equal to the branch key in the parent and to the high fence key in the foster child, whereas the foster key is equal to the low fence key in the foster child. The essential difference to B^{link}-trees [52] is that a foster relationship is resolved by moving the pointer and branch key from the foster parent to the permanent parent, whereas they are copied in a B^{link}-tree.

The original design of b-trees split full nodes precisely in the center in order to ensure at least 50% utilization in each node at all times. A split near the center is usually good enough for practical needs, permitting multiple heuristics for split points. One such heuristic minimizes the size of branch keys. Instead of splitting a full leaf node at 50%, the shortest key that separates the key values at, say, 40% and 60% is posted in the parent node and used to distribute keys among the old and new leaf nodes [5]. Note that suffix truncation can be applied only when splitting leaf nodes; applying it again when splitting branch nodes might guide future root-to-leaf traversals to the wrong cousin. A refinement of this heuristic widens the range, say 30%–70%, in order to avoid cutting a list of duplicate key values. Another heuristic, applied during insertion of a sorted stream of new key values, splits a leaf at the insertion point; a variant of this heuristic splits a full leaf in three, with the center node empty and available for efficient future insertions. During index creation from a sorted stream, as well as during large insertions of pre-sorted key values, the range for choosing a new split key might be 80%–100% if the desired overall storage utilization of 90%. Another technique for multiple sorted insertions creates multiple ghost records at a time, to be filled in with minimal effort during subsequent insertions.

The traditional rigid policies of splitting at precisely 50% or 90% can have an undesirable consequence. If the key value distribution of future insertions mirrors the key value distribution in the initial b-tree, node splits will occur not at a steady rate but in waves [23]. Each such wave amplifies contention in the buffer pool, contention for I/O devices, contention on the data structure for free-space management, and contention for the log buffer and the logging device. Simple policies substantially weaken these waves of misery after index creation, for example splitting consecutive nodes rigidly at 85%, 95%, 86%, 94%,

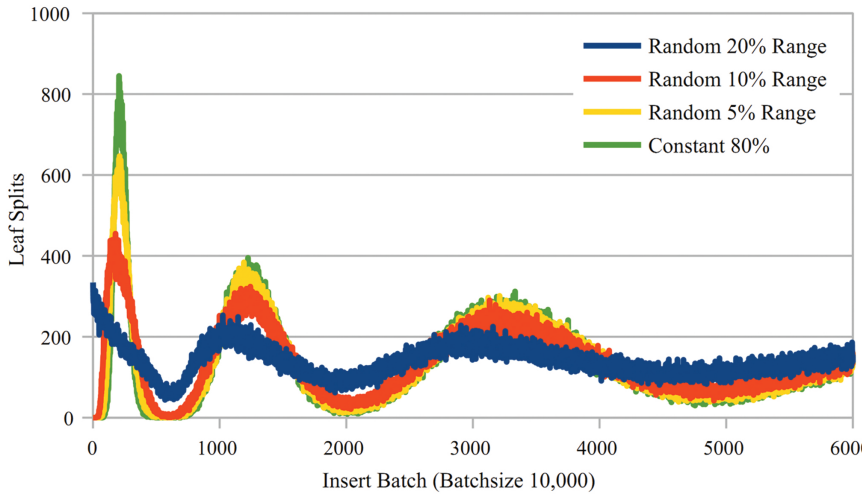

Figure 3.4: Leaf splits after index creation.

etc. or any similar sequence. Policies can be combined, e.g., splitting consecutive nodes at 80%–90%, 90%–100%, 81%–91%, 89%–99%, etc.

Figure 3.4, copied from [23], shows the frequency of leaf splits due to insertions following index creation. Leaving free space during index creation, e.g., filling leaves only to 80%, delays the onset of waves but does not prevent or weaken them much. Randomly changing the fill factor, e.g., using 80% ± 10%, significantly reduces the bursts of system load due to index restructuring. Starting with some leaf nodes immediately filled to maximum capacity, i.e., using 80% ± 20%, requires some leaf splits even during early insertions. Without any delay, the waves of leaf splits are much weaker and much less disruptive. Of course, if the long-term space utilization tends towards 70%, as is common without occasional defragmentation, then index creation might as well start with an average utilization of 70%, e.g., 70% ± 30% utilization in each b-tree node.

In summary of Section 3.3:

- Free space left during index creation can absorb future updates efficiently but increases the index size.

- Temporary local overflow pages absorb updates efficiently but increase the maximum root-to-leaf distance.

3.4 Root-to-Leaf Traversals

A buffer pool and its page movements can be a special case for exploiting a single pointer per node: while both a parent node and a child node are cached in a buffer pool, the parent-to-child pointer can be the virtual memory address of the child or of its descriptor within the buffer pool [42]. With this simple trick, even very large databases can perform like in-memory databases if the application's working set fits within the buffer pool. If buffer pool eviction is simplified using in-memory child-to-parent pointers, splitting a branch node requires some extra in-memory effort.

Another optimization for repeated root-to-leaf traversals caches for each tree level a virtual-memory pointer for the buffer frame, the PageLSN (or version number) of the page, and the slot number chosen in the prior root-to-leaf path. In a foster b-tree, this information is required along the entire root-to-leaf path, even along chains of foster parents and children. This information is useful in scans that need to advance from one leaf to the next, whether that next leaf is a foster child, a sibling (same parent node), or a cousin (shared ancestor but not a shared parent node).

Retracing a root-to-leaf path can be very fast with this information. Pinning pages within the buffer pool can be integrated with latching those pages in shared mode. In fact, it might be possible to employ hardware support in the form of transactional memory. Such pinning and latching should also permit simple and efficient node removal, in particular in b-tree variants such as foster b-trees with a single pointer per node. Similarly, advancing to the next leaf node can be very fast.

Active research includes efficient indexing for correlated attributes [72], e.g., multiple "date" or "money" columns. A b-tree on one column can support correlated columns by adding to each parent-to-child pointer the minimum and maximum of the correlated columns. This can be either minimum and maximum of their values or of their difference to the first column. For the Lineitem table of TPC-H, for example, a b-tree

on "ship date" could add minimum and maximum values of "receipt date" to each parent-to-child pointer. Alternatively, there could be minimum and maximum values of the difference (interval) between ship date and receipt date. It seems that in many cases interval information would be needed only in the b-tree's root node since the minimum and maximum differences would be nearly constant within the entire b-tree. While correlation is frequent among "date" columns and common among "money" columns, the technique supports any set of correlated columns. In effect, it adds the functionality of zone maps and zone filters [36] to b-tree indexes but it exploits the hierarchical structure of b-trees. As in other zone filters, minimum and maximum information can be augmented with a bit vector filter in order to support equality queries or with second-to-minimum and second-to-maximum key values in order to mitigate the effects of outliers. In addition to traditional b-tree indexes, zone filters are promising for column storage in b-tree structures [28].

Figure 3.5 shows two alternative formats for entries in branch nodes. On the left, a traditional entry pairs a key value, e.g., a ship date of the Lineitem table of TPC-H, and a parent-to-child pointer, e.g., a page identifier on storage or an address in memory. On the right, an extended entry adds the minimum and maximum difference to a correlated column, e.g., the interval from ship date to receipt date. A single b-tree might employ both of these formats. In this specific example, the extended format might be used only in the root node; lower branch nodes have no need to refine the bounds indicated in the root node. In other examples, the root node might not have tight, useful bounds but lower branch nodes do. In a further variant, a lower branch node might overwrite only one of minimum and maximum. In the example here, the minimum interval from ship date to receipt date might always be zero days but the maximum interval might differ among leaf nodes and their entries in their parent nodes. Finally, a single b-tree might support efficient search on even more than two correlated columns. For example, for

Branch key	Pointer
Ship date	Page identifier

Branch key	Pointer	Minimum	Maximum
Ship date	Page identifier	Interval of receipt dates	

Figure 3.5: Alternative formats for branch node entries.

the Lineitem table of TPC-H, a single b-tree on "ship date" might include minimum and maximum information for both "commit date" and "receipt date." And coming back to the techniques of Section 5, such a b-tree also supports efficient scans sorted on any one of these correlated columns.

In summary of Section 3.4:

- With swizzled parent-to-child pointers, b-trees in a buffer pool perform like in-memory indexes.
- Sequences of repeated or related searches can benefit from cached root-to-leaf paths.
- A single b-tree can index one attribute precisely and a correlated attribute quite effectively using minimum and maximum information – time and money information within the same application is often highly correlated. Differences instead of values often enable compression.

3.5 Self-Repairing B-Trees

Effective repair requires both diagnosis of inconsistent state and recovery of consistent information [39]. Self-repairing b-trees perform comprehensive consistency checks during index search and invoke efficient single-page repair if an error is found. Optionally, the consistency checks can run not only during pre-deployment quality assurance but also after deployment. In both cases, they must be incremental. Write-optimized b-trees and foster b-trees enable such self-testing because each piece of information is stored twice, e.g., as branch keys and as fence keys. The only important information stored only once is the PageLSN value, which might be copied to a parent node whenever a newly modified child is evicted from the buffer pool and written to storage.

Efficient single-page repair [34], [38] relies on log records in the recovery log and in the log archive, plus an out-of-date page image in the most recent database backup. Each log record includes the replaced PageLSN value, i.e., a pointer to the preceding log record for the same database page. If a b-tree node is found inconsistent, the expected

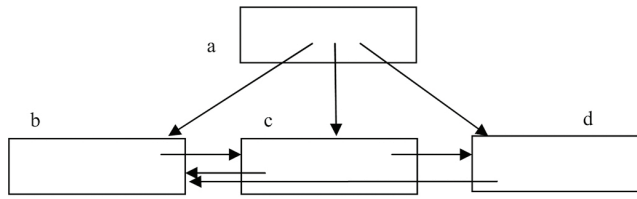

Figure 3.6: An incomplete leaf split.

PageLSN value in the parent node anchors a linked list of log records required to recover the child node by direct access to the required log records. Suitable mechanisms for compressed log archives and database backups exist [34].

Figure 3.6 shows the result of splitting a leaf node incorrectly. When leaf node b was split and leaf node c was created, the backward pointer in successor node d incorrectly remained unchanged. A subsequent (descending) scan of the leaf level will produce a wrong query result, and subsequent split and merge operations will create further havoc. The problem might arise after an incomplete execution, an incomplete recovery, or an incomplete replication of the split operation. The cause might be a defect in the database software, e.g., in the buffer pool management, or in the storage management software, e.g., in snapshot or version management. In other words, there are many software modules with many thousands of lines of code that may contain a defect that leads to a situation like the one illustrated in Figure 3.6. Different b-tree inconsistencies result if all nodes but c are saved correctly in the database; in that case, a, b, and d point to a page full of garbage. If all nodes but b are saved correctly, the key range in b would not conform to the separator keys in a. If all nodes but a are saved correctly, a search for records in node c will not find them. B-tree verification must find such errors reliably and as efficiently as possible.

Figure 3.7 shows the logic that runs when a database page comes into the buffer pool and that invokes single-page repair if necessary and available.

Figure 3.8 shows the logic of single-page repair. A self-repairing index employs the page recovery index (one per device) only for the b-tree root page. For any other b-tree node, the parent node contains

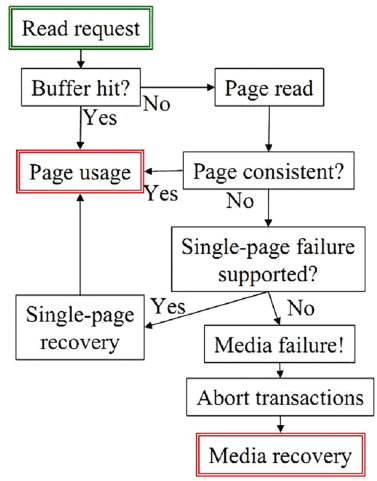

Figure 3.7: Page retrieval logic.

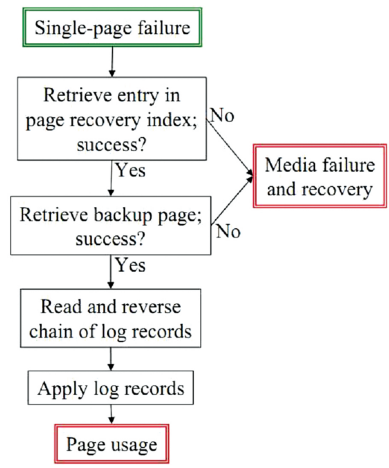

Figure 3.8: Single-page recovery logic.

the required information, namely the log sequence number of the most recent update.

In summary of Section 3.5:

- Redundant information in b-trees enables continuous comprehensive consistency check as well as efficient localize repair using the database recovery log.

3.6 Deferred Updates and Deferred Index Optimization

A different approach to fast b-tree insertions builds on physical data independence and automatic maintenance of secondary indexes. In other words, this approach fits into database management systems more than into key-value stores. In particular, insertions may be captured only in a table's primary index but not in the secondary indexes. Updates may be mapped to insertions of replacement records and deletions to insertions of tombstone records – version records may be useful for read-only transactions in snapshot isolation [9] and "time travel," i.e., queries of past database contents. For maximum capture bandwidth, initial insertions into the primary index merely create memory-size runs within a partitioned b-tree. Subsequent reorganization and optimization of the primary index not only merges partitions but also extracts index entries for all secondary indexes and appends them there as sorted runs. Extraction, propagation, reorganization, and optimization may be continuous processes, possibly even running on demand in preparation of or even as side effects of query execution. Reorganization and optimization may include garbage collection, i.e., removal of out-of-date version records.

Figure 3.9 illustrates deferred update propagation when both primary and secondary index are partitioned b-trees. Insertions are appended as new partitions in the primary index. Reorganization and optimization of the primary index extracts, sorts, and appends new

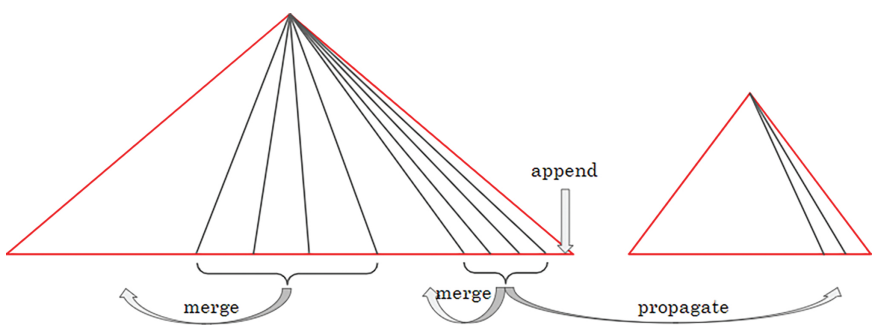

Figure 3.9: Deferred update propagation using partitioned b-trees.

partitions to the secondary index. Subsequent reorganization and optimization of the secondary index brings the database into a steady state with fully optimized, single-partition indexes. Between the initial insertions and update propagation, a query searching the secondary index must also scan the new partitions of the primary index, i.e., those not yet propagated to the secondary index.

These ideas and techniques apply not only to secondary indexes of tables but also to secondary indexes of materialized views. In fact, deferred update propagation even applies to tables and their materialized views. In that case, a query that searches in a materialized view (and its indexes) must also search in the deferred updates of all base tables. An alternative design propagates all deferred updates before executing a user query. In other words, update propagation occurs either in background activities or in preparation of user queries. Both designs can use system transactions or utilities transactions outside of user transactions. Update propagation as side effect within a user's query evaluation plan complicates transaction management, e.g., in cases a user query stalls, fails, or aborts.

While deferred update propagation from a primary index to its secondary indexes is rarely implemented, deferred optimization of storage structures is common in the forms of log-structured merge-trees [63] or stepped-merge forests [47], the topic of the next section.

In summary of Section 3.6:

- With updates mapped to insertion of replacement records and deletions mapped to insertion of tombstone records, speeding up all kinds of updates means speeding up insertions.
- New insertions may be captured in a table's primary storage structure only, letting secondary indexes fall behind and catch up eventually.
- Partitioned b-trees can separate new insertions from old data – initial data goes into new partitions, eventual reorganization and partition merging can, as a side effect, add new partitions to dependent storage structures such as secondary indexes and materialized views.

- Queries can combine out-of-date secondary indexes and recent insertions in primary storage structures, or force catch-up as a preparation step, or compute and apply catch-up as side effect.

3.7 Summary of Tree Structure Improvements

In summary, many improvements to b-tree structures have been proposed. Those pertain both to search performance and, perhaps more importantly, to operational characteristics such as mean time to repair, high-bandwidth updates, and more.

4

Insertion-Optimized B-Trees

Insertion-optimized b-trees often are the key components of stream indexing and many key-value stores. There seem to be two alternative approaches to designing insertion-optimized b-trees. The common design approach starts with an efficient in-memory index (a b-tree, a red-black tree, or any other ordered tree structure) and extends it to spill data to a b-tree on temporary or persistent storage. For example, log-structured merge-trees scan the memory contents in key value order, spill as required to create free index nodes for newly arriving contents, immediately merge the spilled contents with the b-tree on storage, write the merge result as a new on-storage b-tree, and replace the old on-storage b-tree when the new one is complete.

Repeatedly merging new input data from memory into (or with) an existing b-tree ensures that queries search only two indexes, one index in memory and one b-tree on storage. However, since this process has O (N^2) complexity, it does not scale endlessly. In order to limit write amplification, i.e., the number times each index entry is written in multiple reorganization steps, some designs occasionally start over with an empty on-storage b-tree and eventually merge multiple on-storage b-trees to form a single large b-tree. Repeated merging slowly grows this

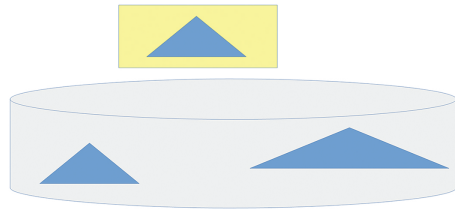

Figure 4.1: The first merge step in an LSM-forest.

b-tree until this second O (N^2) process requires starting over again. In other words, log-structured merge-forests mix merging new data within a level and merging from one level to the next.

Figure 4.1 illustrates the first merge step in a log-structured merge-forest. On top is an in-memory b-tree. When memory first overflows, its contents spill and create the on-storage b-tree on the left. Its size is equal to the b-tree in memory (or twice if the spilling logic resembles replacement selection). When memory overflows again, memory contents and the on-storage b-tree are merged to form the on-storage b-tree on the right.

Figure 4.2 illustrates the second merge step in a log-structured merge-forest. The spilling memory contents are merged with the on-storage b-tree twice the size of memory (on the left), forming the on-storage b-tree three times the size of memory (on the right). With each merge step, the b-tree size grows by the size of the b-tree in memory.

Figure 4.3 illustrates a merge strategy that mixes merge steps typical for log-structured merge-forests with merge steps typical for external merge sort. The former are characterized by additive increases in run sizes, i.e., one more unit is added to an existing run of multiple units,

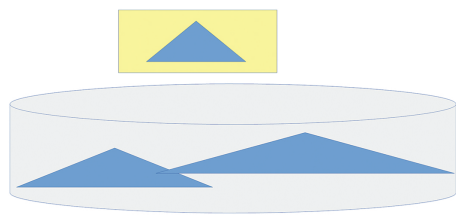

Figure 4.2: The second merge step in an LSM-forest.

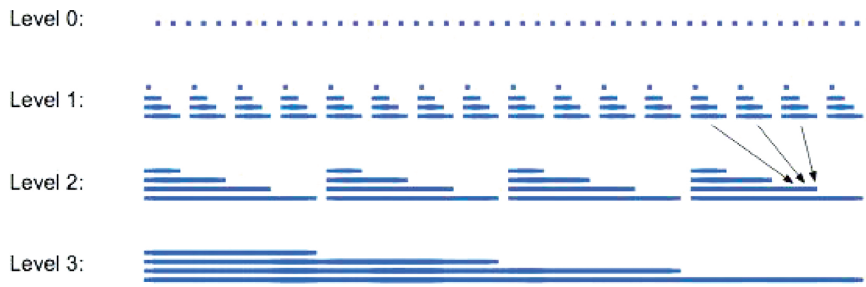

Figure 4.3: A mixed merge strategy.

whereas the latter are characterized by multiplicative increases in run sizes, i.e., multiple runs of similar or equal sizes are merged to form a run of much larger size. The arrows from level 1 to level 2 show such a merge step.

An alternative design approach starts with an external merge sort and modifies it only inasmuch as required to enable indexed search, e.g., storing runs as b-trees or as partitions in a partitioned b-tree, organizing runs in levels, merging runs from one level to the next, maximizing merge fan-in, etc. While the merge fan-in of a traditional external merge sort is limited by the memory available for input buffers (divided by the page size), the merge fan-in here is usually limited by the number of runs existing at a time, i.e., the query effort for searching all runs. This is the basic approach that led to the design of stepped-merge forests [47].

In summary of Section 4 so far:

- With b-tree creation routinely sped up by first sorting the future index entries, insertions can run as never-ending index creation.
- Log-structured merge-forests and their variants may be seen as never-ending external merge sort with occasional merging and b-trees as the run format.

4.1 Tradeoffs in Insertion-Optimized B-Trees

Both designs enable queries over recently inserted index entries. If the in-memory index has suitable concurrency control, e.g., orthogonal key-value locking in a b-tree, queries can search through new insertions as soon as those are committed. In order to limit the number of b-tree partitions each query must search, merging eagerly with small fan-in seems best. On the other hand, in order to minimize the number of merge steps and the overall merge effort, a large merge fan-in seems best. Thus, there is a tradeoff between merge efficiency and query performance.

Not every partition contains information about each key value in the domain. Many queries can avoid many b-tree traversals if bit vector filters for each partition summarize its contents. These bit vector filters can be large and require substantial memory. Thus, there is a tradeoff between allocating memory for bit vector filtering or for buffering index nodes.

In order to guarantee satisfactory query performance, it may be useful to exclude the most recent insertions (and thus the smallest b-trees) from queries. This creates another tradeoff dimension, which might be called the freshness of query results.

Figure 4.4 illustrates the resulting tradeoff between (i) data freshness or information latency from initial capture to first query, (ii) effort for information capture and indexing including merging runs (deltas, partitions), and (iii) effort for querying available information using the currently existing indexes. Each of the points in this triangle has its own optimization levers, but tension between them and the need for tradeoffs remains nonetheless.

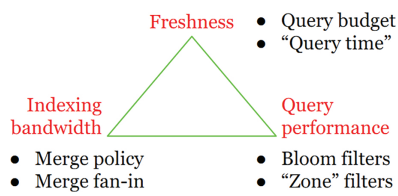

Figure 4.4: Tradeoffs in log-structured merge-forests.

In summary of Section 4.1:

- Traditional external merge sort, e.g., during index creation, is efficient with a large merge fan-in.
- In log-structured merge-forests, efficient merging means many partitions waiting to be merged, but efficient queries require eager merging with a small fan-in.
- This trade-off can be alleviated by producing query results not entirely up-to-date, i.e., ignoring the most recent and thus smallest partitions.

4.2 Merge Optimizations

Optimizing the merge effort includes two directions. First, the O (N^2) components may be replaced by O (N log N) algorithms; and second, the constants in O (N log N) algorithms can be improved. Both directions attempt to reduce or limit the number of times a data record is merged.

The first one of these directions accelerates the switch from merging new data into (or with) an existing b-tree to merging data strictly in levels like a traditional external merge sort. Put differently, a log-structured merge-tree initially uses only one on-storage b-tree, repeatedly merging memory contents into (or with) this b-tree. Thus, the oldest index entries are merged repeatedly with new contents. This is the cause of the O (N^2) behavior. In contrast, a perfectly balanced strategy always merges index entries that all have gone through the same number of prior merge steps. For example, index entries (runs, deltas, index partitions) that have already been merged once are merged only with other index entries (runs, deltas, index partitions) that have already been merged once; the output of that merge step will be merged only with other index entries (runs, deltas, index partitions) that have already been merged twice; etc. The result is a classic O (N log N) algorithm.

The second one of these directions reduces the merge levels by increasing the merge fan-in. In a traditional external merge sort, the merge fan-in is limited by memory size, page size (unit of I/O), and buffer requirements for asynchronous read-ahead and write-behind (e.g.,

double buffering vs. forecasting). In log-structured merge-trees, the fan-in is limited by the desire for efficient queries, i.e., few existing runs (deltas, partitions) at a time, but a more efficient merge strategy uses a higher fan-in.

Both of these directions increase the number of runs (partitions) present on storage at any point in time. Thus, reducing the merge effort increases the effort required in each query and must be counter-balanced by query optimizations.

Figure 4.5, copied from [67] including the original labels and the explanatory caption, illustrates considerations and difficulties in scheduling merge steps in most designs for insertion-optimized forests with many b-trees. For example, if a greedy policy chooses a key range for the next merge step that frees the most memory with the least I/O, subsequent merge steps cannot be as efficient – while the greedy policy is efficient immediately, it is perhaps less efficient in the long run. Similarly, if one of the merge levels is permitted to fall behind, it is possible that a situation arises in which no further input can be absorbed, i.e., the stream index fails one of its basic requirements.

In summary of Section 4.2:

- In log-structured merge-forests, as in traditional external merge sort, balanced merging enables a classic O (N log N) algorithm.
- Merging with a large fan-in reduces the count of required merge levels but forces queries to search more partitions waiting to be merged.

4.3 Search Optimizations

Query performance is determined by the efficiency of search within a partition, the number of partitions in existence, and the number of partitions where search can be avoided. Efficient search within a b-tree or within a partition is discussed later. The number of partitions in existence is a consequence of the merge strategy, as discussed earlier. The number of partitions where search can be avoided depends on auxiliary data structures. Bit vector filters are most commonly used

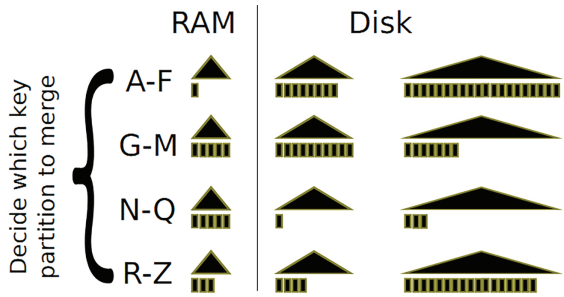

Figure 3: ***Partition schedulers*** **leverage skew to improve throughput. A greedy policy would merge N – Q; which uses little I/O to free a lot of RAM.**

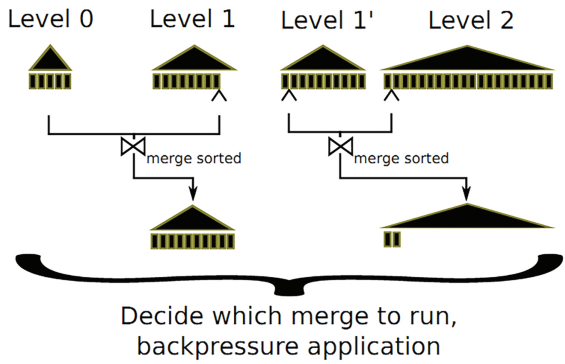

Figure 4: ***Level schedulers*** **(such as spring and gear) decide which level to merge next. This tree is in danger of unplanned downtime: Level 0 (RAM) and 1 are almost full, and the merge between level 1 and 2 has fallen behind.**

Figure 4.5: Merge scheduling.

for this purpose, possibly augmented with information about minimum and maximum values.

The effectiveness of bit vector filtering depends on the sizes of the bit vector filter, the key domain (distinct possible key values), the population (actual values), and the key value distribution (e.g., many duplicate key values). If key domain and key value distribution are

given and the population equals the size of individual partitions, the principal tuning opportunity is the size of bit vector filters. In large partitions, a traditional search is fairly efficient relative to the data size, whereas searching many small partitions can be expensive, futile, and wasteful. Thus, some researchers suggest focusing bit vector filtering on small partitions (runs, deltas) in a log-structured merge-forest [63], i.e., adjusting sizes of bit vector filters according to the sizes of individual partitions.

An entirely different way to avoid searching many small partitions modifies the query semantics, as discussed earlier as one of the tradeoffs in stream indexing. Instead of querying all committed input data, a system design might limit query execution to a fixed number of partitions (starting with the oldest one) or to a given maturity level (e.g., search only partitions in and above the second merge level). Of course, this gives up on the near-real-time promise of stream indexing. It might be practical to adopt such a policy adaptively, e.g., apply it only during times of high system load.

Figure 4.6 shows the effect of querying only old and thus large partitions. The smallest partitions cover only a single minute and might be formed within an in-memory workspace. Older partitions are the result of merging smaller partitions. In this example, the merge fan-in

3 months	Jan 1 – Mar 31	
	Apr 1 – Jun 30	
	Jul 1 – Sep 30	6×
½ month	Oct 1 – Oct 15	
	Oct 16 – Oct 31	
	Nov 1 – Nov 15	
2 days	Nov 16 – Nov 17	7-8×
6 hours	0m – 6am	8×
	6am – 12n	
	12n – 6pm	
1 hour	6 – 7pm	6×
7-8 minutes	7:00 – 7:08pm	8×
	7:08 – 7:15pm	
1 minute	7:15pm	7-8×
	7:16pm	
	7:17pm	
	7:18pm	

Figure 4.6: Querying only the oldest partitions.

is consistently in the range 6–8, as indicated on the right of Figure 4.6. A query system might support queries for data in partitions of an hour or larger, as shaded in Figure 4.6. On one hand, query performance improves because there are only few partitions to search. On the other hand, query results are based on out-of-date information and index contents. A possible compromise policy starts with a given query budget, e.g., 20 partitions, and searches as many partitions as possible within the budget, from oldest and largest partition towards newest and smallest partition.

In summary of Section 4.3:

- Queries can reduce the count of partitions to search if each partition has a bit vector filter or if queries may ignore some time window, e.g., the most recent insertions still lingering in small partitions waiting to be merged.

4.4 Storage Structures

An insertion-optimized b-tree consists not of a single b-tree but of many b-trees, one for each partition (delta, run). Thus, it should really be described as a forest, not a tree. Creation and removal of individual trees requires some form of metadata or catalog. If each tree is an individual file within a file system, creation and removal of trees is tracked by the file system and the entire system relies on the file system's support for concurrency control, recovery, and high availability.

An alternative design employs a single b-tree but prefixes the user-defined key with a partition identifier. Except for this prefix, a partitioned b-tree can be a standard b-tree implementation with its usual concurrency control, logging, recovery, etc. Such a partitioned b-tree does not require an external facility to track which partitions currently exists or which partitions a specific query (with its given snapshot isolation timestamp) must search. Creation of a new partition is implicit in insertion of a first index entry with a new partition identifier; deletion of a partition is implicit in removal of the last index entry (or all index entries) with a specific partition identifier. Search in this storage structure enumerates existing partition identifiers and searches each

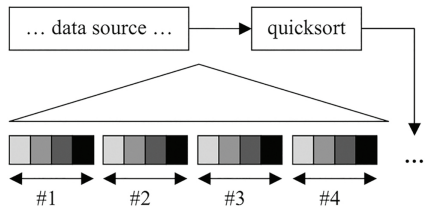

Figure 4.7: A partitioned b-tree.

partition, very much like the "multi-dimensional" access method [53] for multi-column b-trees.

Figure 4.7, copied from [37], shows a partitioned b-tree. The labels at the bottom are partition identifiers. These key prefixes in each index entry require no storage space if the b-tree employs prefix truncation [5]. Continuous loading adds new partitions, reminiscent of the run generation phase within an external merge sort.

In summary of Section 4.4:

- Partitioned b-trees use an artificial leading key field for a partition identifier; with that, a single b-tree can hold many partitions of a log-structured merge-forest.
- Prefix truncation eliminates practically all storage cost for these partition identifiers.
- "Multi-dimensional" b-tree search ("MDAM") enumerates and searches partitions.

4.5 Continuous Merging with Staggered Key Ranges

Like traditional external merge sort, all prior designs for log-structured merge-forests and stepped-merge forests invoke merge steps on demand. The overall result can be waves of merging, common in today's implementations of log-structured merge-forests. An alternative approach merges in multiple levels just like external merge sort but each merge level is active all the time. If each merge step progresses with a bandwidth matching the data ingestion bandwidth, then the data volume present at each merge level remains constant.

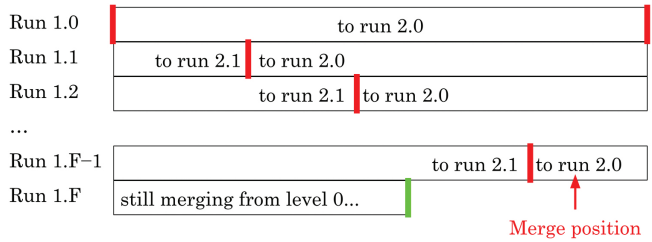

Figure 4.8: Staggered key ranges in continuous merging.

The principal new idea is to let a merge step add an input run at any time and at any key value, specifically at the key value determined by the current merge progress. Each run contributes its upper key range to one merge step and its lower key range to the next merge step. A run contributing to a single merge step from the minimum to the maximum key value is possible but it is the exception, not the rule as in traditional external merge sort and traditional log-structured merge-forests.

Figure 4.8 illustrates continuous merging with staggered key ranges. The key domain stretches from left to right. The diagram shows only one merge step, from level 1 to level 2, with merge fan-in F. Each run at level 1 (except run 1.0 and run 1.F) contributes to two runs at level 2. Merging starts at a different key value K for each run at level 1, indicated by red bars in Figure 4.8. Key values greater than K contribute to one run at level 2 (here run 2.0) and key values smaller than K contribute to the next run at level 2 (here run 2.1). The current merge position (bottom right) indicates that the level-1-to-level-2 merge is about to finish run 2.o. When this merge step finishes and the level-1-to-level-2 merge is ready to start creating run 2.1, the level-0-to-level-1 merge will finish writing run 1.F. The merge into run 2.1 will include none of run 1.0 and all of run 1.F. As the merge step creating run 2.1 progresses through the entire key domain, it will drop runs 1.1, 1.2, etc. and add runs $1.F + 1$, $1.F + 2$, etc. so it merges F runs at all times.

The principal advantage of the new continuous merge schedule is that runs never wait from writing to reading (merging). For example, immediately after run generation finishes a new level-0 run, the active level-0-to-level-1 merge increases its fan-in and adds the new run to its

inputs, starting at the key value equal to the current merge progress. The same applies when one merge level writes a run and the next merge level immediately starts reading it. A more aggressive merge strategy adds a run when, within that run, the writer "overtakes" the reader, i.e., the current key value of the writer is larger than the current key value of the next merge step.

The current storage requirements on its input side pace each merge step. A merge step proceeds if and only if there is more input than there should be, which is governed by the desired merge fan-in and the merge level. For example, for level-0 runs, the total size of their unmerged key values should equal the product of desired merge fan-in and the size of the in-memory workspace allocated for run generation, which is also equal to the expected run size at level 1. The level-0-to-level-1 merge proceeds whenever and while there is more unmerged data on level 0 than this threshold. For runs at level-l+1, this sum should be larger than for level-l by a factor equal to the desired merge fan-in.

When necessary, continuous merging can absorb bursts of input. In the simplest case, a burst of input will briefly create more level-0 data than desired in the steady state. In that case, merging will resume and promote data into higher merge levels. When a merge step falls behind, e.g., due to a burst of input, the merge fan-in may temporarily be higher than the desired merge fan-in. In a more complex case, the processing load of transforming unsorted input into level-0 runs limits the processing bandwidth of the merge steps; in that case, data will accumulate in level 0 but after the input burst subsides, merging will promote data until achieving a steady state with the desired sizes per level. If bursts of input can be anticipated, merging can over-eagerly promote data, reduce the data volume per merge level, and thus create extra space to absorb an input burst gracefully.

In external merge sort, the memory size limits the merge fan-in, whereas in continuous merging and log-structured merge-forests, achieving acceptable query performance forces low fan-in in all compaction merge steps. Query execution gathers and combines data from all existing runs, possibly tempered by bit vector filtering, but typically with a much higher fan-in than the fan-in desired in continuous merging. With all runs stored as b-trees or b-tree partitions, a query may merge

all data from all runs in a single merge step and, as a side effect of query execution, it may deposit the data in a high-level run, i.e., where continuous merging would eventually have written it, and erase its input data in the runs from which it read and merged. Thus, adaptive merging [37] was originally designed only for partitioned b-trees but it can also aid continuous merging.

Not only the merge strategy but also the storage structures bear improvement. Instead of a forest of many b-trees, with frequent updates in a catalog or a file system to track the current set of b-trees, a single partitioned b-tree can represent many runs (partitions, deltas). Efficiently writing a run as a b-tree requires buffer space for each level of the b-tree, from root to leaves, and querying a partitioned b-tree requires a root-to-leaf traversal within each partition. In contrast, a linear partitioned b-tree avoids the upper tree levels (also known as branch pages). Instead, each partition holds key-pointer pairs (also known as branch entries) of its preceding partition. Each data page in a partition requires one key-pointer pair in the succeeding partition. In addition, the low fence key in each data page holds a pointer to a data page in the preceding partition. Thus, a search for a specific key value can access one data page in each partition, very much like a linked list, avoiding root-to-leaf traversals and binary search in all but the in-memory partition.

Figure 4.9 shows a linear partitioned b-tree with an in-memory index (green triangle), files of sorted pages on persistent storage (blue rectangles), and page pointers embedded in both the in-memory index and the sorted files (red arrows). The sorted pages (blue rectangles) are very much like leaf nodes in a traditional b-tree index. The number of page pointers in the in-memory index equals the count of data pages in the most recent file on persistent storage. The number of files on persistent storage can be smaller or larger than two. The number of

Figure 4.9: A linear partitioned b-tree.

page pointers in one persistent file equals the count of pages in the preceding persistent file plus the count of pages in the current persistent file. Put differently, each data page has one incoming pointer that serves the role of the parent-to-child pointer in a traditional b-tree, plus one outgoing pointer attached to low fence key of each data page.

Data access in a linear partitioned b-tree is very similar to a linked list. It is not possible to skip over a partition, even if a bit vector filter might indicate that a given search key does not exist there [14]. Thus, some gains in search efficiency that are possible in both log-structured merge-trees [63] and stepped-merge trees [47] are not possible in a linear partitioned b-tree.

A near-linear partitioned b-tree eliminates most of this disadvantage. Instead of each partition referencing data pages in its immediate predecessor, only some of the partitions contain backward references. For example, instead of long linked lists as illustrated in Figure 4.9, only every nth partition references earlier data pages, but it covers n preceding partitions rather than just one. Thus, the in-memory index gathers and retains page references while writing $n - 1$ partitions and saves all of them in the nth partition. Data access in a near-linear partitioned b-tree must visit every nth partition but might skip $n - 1$ of every n partitions based on bit vector filtering.

As illustrated in Figure 4.9, the key-pointer pairs of the most recent partition remain in memory. While writing this storage structure, the same ordered in-memory index can serve run generation, i.e., hold data records, and data access, i.e., guide queries to the correct data page in the most recent persistent partition. In fact, this in-memory index can anchor multiple linear partitioned b-trees, one per merge level. If near-linear partitioned b-trees are employed, the in-memory index anchors multiple merge levels and multiple run per merge level.

Figure 4.10 illustrates two linear partitioned b-trees for two merge levels managed by a single in-memory index, e.g., an in-memory b-tree. The in-memory index interleaves two types of index entries, data records and pointers into the most recent deltas (partitions, runs) on storage. These pointer records indicate a merge level such that queries and merge steps can search and merge the correct data pages. More than

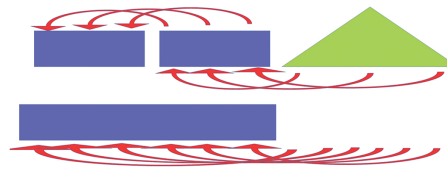

Figure 4.10: Linear partitioned b-trees for continuous merging.

two merge levels are possible, depending on the data volume, the size of initial deltas (runs, partitions), and the merge fan-in.

For read-write transactions and their concurrency control, the in-memory partition may serve not only as workspace for data ingestion and run generation but also as cache for active transactions. Thus, this design augments the prior one. When a read-write transaction reads a key value, it typically must retrieve or even derive the most recent version from multiple recent partitions. The new idea is to have this transaction (or a system transaction) insert this most recent version just as if it were a new version. The destination of this insertion is, of course, the in-memory partition. Run generation and spilling from memory must skip over locked key values just as it ought to skip over index entries that absorb many new insertions in an index with aggregation of values or versions. Thus, the in-memory buffer pool serves as a combined cache for ingestion, queries, and updates; and orthogonal key-value locking can provide a fine granularity of locking with minimal false conflicts.

In summary of Section 4.5:

- Continuous merging is a multi-level merge strategy that merges at all levels at all times, guided and driven by the data volume waiting to be merged at each level.
- Linear partitioned b-trees avoid traditional branch nodes by interleaving one b-tree's branch entries (key-pointer pairs) in the next b-tree's leaf nodes (data pages); an in-memory b-tree anchors linked lists in multiple merge levels.
- Continuous merging in linear partitioned b-trees uses the in-memory b-tree to buffer and to index recent insertions, for run generation, to manage the merge progress concurrently in multiple

merge levels, and to free data pages rendered obsolete by the merge progress.

4.6 Summary of Insertion-Optimized B-Trees

In summary, log-structured merge-forests and stepped-merge forests are ubiquitous in data warehousing, stream indexing, log analysis, and more. Continuing research keeps finding new opportunities for optimizations and better tradeoffs between capture bandwidth and indexing effort, query effort and performance, and data freshness or information latency. Continuous merging with staggered key ranges is based on external merge sort and is unique because it merges sorted runs not from start to finish but in the fashion of a merry-go-round.

5

Query Processing

Query execution engines are the context for b-trees not only in database management systems but also in general data processing systems and dataflow environments such as MapReduce [15], [16], its derivatives, and its replacements. Query processing permits information derivation and extraction beyond simple filtered and ordered retrieval of data. Query processing techniques also contribute to efficient maintenance of b-tree indexes, e.g., in index-by-index update plans [1] and in adaptive merging [37], including adaptive merging in insertion-optimized b-tree forests such as stepped-merge forests [47] and in continuous merging with staggered key ranges. The present section reviews some relevant recent techniques in more detail.

5.1 Grouping and Aggregation During Run Generation

Traditional sort-based algorithms for duplicate removal, grouping, and aggregation avoid all duplicate key values in all runs on temporary storage [7]. This in-sort grouping can provide a substantial performance improvement over sorting with subsequent in-stream grouping. The algorithm and data structure for run generation does not matter: both

read-sort-write cycles using quicksort and replacement selection using a priority queue enable in-sort grouping.

If the output size of duplicate removal, grouping, and aggregation fits in memory, however, another optimization provides another substantial performance boost. Instead of quicksort or priority queue, run generation can employ an in-memory ordered index, e.g., an in-memory b-tree. The indexing functionality permits absorbing duplicate key value as they are read from the input, not just before they are written to runs on temporary storage. Thus, if the operation's output fits in the available memory allocation, run generation using an in-memory b-tree index avoids all temporary storage.

At the same time, the ordering of the in-memory b-tree index can serve run generation. An eviction scan can free memory as needed by flushing key values from memory to temporary storage. The functionality of replacement selection does not even require marking key values for the current or the next run, in particular the effect of runs twice as large as the available memory allocation for random input and much larger for somewhat pre-sorted input.

Figure 5.1 illustrates run generation using an ordered in-memory index. The index continuously grows due to insertions of rows and key values from the unsorted input. New key values create new index entries; key values equal to ones already in the index are absorbed by aggregation. In the ideal case, the entire input can be absorbed within memory. Otherwise, a scan thread continuously evicts leaf pages and writes them to temporary storage to add them to a forest (of many trees) or a partitioned b-tree (of many partitions).

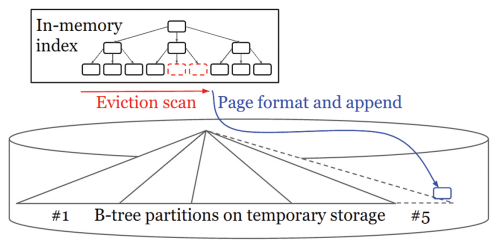

Figure 5.1: Run generation using an ordered in-memory index.

In summary of Section 5.1:

- Traditional techniques for in-sort duplicate removal, grouping, and aggregation avoid duplicate keys in runs on temporary storage, i.e., while writing runs.
- Run generation using an in-memory b-tree can avoid duplicate keys in memory, i.e., while consuming unsorted input.
- If the final output fits in memory, no runs on temporary storage are required.

5.2 Wide Merging During the Final Merge Step

Complementing the in-memory index for run generation, in-sort grouping and duplicate removal can use an in-memory index for a final merge step with a merge fan-in potentially higher than a traditional merge step with the same memory allocation and page size (unit of I/O). Compared to traditional merging in an external merge sort, wide merging is not limited to a specific fan-in. Instead, wide merging uses its memory allocation for an in-memory index and processes one page at a time from the runs of the aggregation input. Thus, wide merging can be much more efficient than traditional aggregation within sort, e.g., saving an entire intermediate merge level.

Figure 5.2 illustrates the flow of data in wide merging. Using in-memory buffer space for a single page, the ordered in-memory index absorbs all rows and key values from all runs or partitions on a temporary

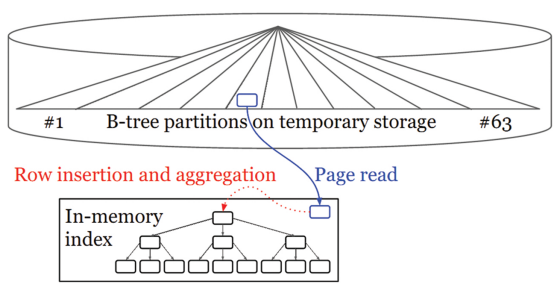

Figure 5.2: Wide merging using an ordered in-memory index.

storage device. The merge logic progresses through the domain of key values as in a very wide merge step. Reflecting this progress, the active key range in the in-memory index turns over continuously. The left edge of the in-memory index produces final output and the right edge adds new key values from the external runs. A priority queue guides the page consumption sequence during wide merging.

In summary of Section 5.2:

- A traditional merge step uses one input buffer per merge input; wide merging uses a single input buffer for all merge inputs.
- After reading a page from a merge input, its data records are aggregated into an in-memory b-tree and the input page discarded.
- The read sequence during wide merging is the same as in traditional merge logic with the same high fan-in.

5.3 Index Intersection

For data warehouse queries, indexing techniques are essential. For example, consider an example query "from T where A in [11, 13, 17, 19] and B between 23 and 29 and $C <> 31$" with secondary indexes on column A, on column B, and on column C. If column A can have duplicate values, and if the index on column A has a sorted list of row identifiers for each distinct value of A, a 4-way union operation and merge algorithm can produce a sorted combined list of row identifiers. If column B has integer values, a 7-way union and merge can do the same for column B using the index on B. The conjunction ("and") requires an intersection operation, computed efficiently by merging the two sorted union results. The index on column C can produce a list of row identifiers satisfying "$C = 31$;" a difference operation and binary merge can efficiently compute the list of row identifiers satisfying the entire query predicate.

The earlier survey of modern b-tree techniques [29] focused on hash join as the principal algorithm for intersection. It turns out, however, that merging is more efficient with respect to memory but also faster for initial results, which is desirable in the context of an "exists" predicate,

for example. Moreover, due to the focus on hash join, the earlier survey also ignored prefix truncation for compression in b-trees and offset-value coding for efficient query execution such as merging.

In summary of Section 5.3:

- Index entries are sorted in b-trees, including the row identifiers in non-unique database indexes.
- Merging sorted lists of row identifiers efficiently computes the intersection for "and" predicates, union for "or" predicates, and difference for "not in" predicates.

5.4 Index Joins

When accessing a database table through a secondary index, the expensive part is random I/O to fetch the selected rows. Some techniques can reduce the cost, e.g., sorting the set of row identifiers. Nonetheless, it is even better to avoid this cost altogether. If a secondary index holds all the columns required for a query, including concurrency control information such as timestamps, random access in the table can be skipped. This is commonly called "index-only retrieval" or an index "covering" a query. Some database systems support including in a secondary index some columns that are neither search keys nor row identifiers. Obvious candidates for this technique are foreign keys and date columns, because many queries "navigate" through a maze of database tables and many business intelligence queries focus on times and dates.

If a single secondary index fails to cover a query, random access in the table seems required. However, in some cases it is more efficient to join two (or more) secondary indexes in order to assemble the required result rows. The join predicate focuses on the row identifier shared among all secondary indexes.

Joining secondary indexes is particularly advantageous if some or all of the participating indexes also support a query predicate, i.e., if a partial scan of the index suffices, and if the join can be very efficient, i.e., an in-memory hash join and even a merge join. A merge join is readily possible if sorted sets of row identifiers can be scanned and merged from each index, as discussed in Section 5.3. In some important

ways, this query execution strategy is similar to assembling rows from columnar storage, even if columnar database designs usually put more emphasis on compression.

In summary of Section 5.4:

- "Index-only retrieval" avoids random I/O when a secondary index "covers" a query.
- Multiple secondary indexes of the same table may cover a query when joined on the shared row identifier.
- B-tree indexes enable efficient merge join plans akin to row assembly from columnar storage.

5.5 Prefix Truncation and Offset-Value Coding

For maximal compression by run-length encoding, columnar databases store their tables sorted. Sorted row stores, e.g., b-trees, can compress leading sort columns by prefix truncation, i.e., storing columns or bytes starting with the first difference to the row immediately preceding. The compression achieved is the same for rows and columns; it also equals the sharing opportunity in tries [18]. Row-by-row prefix truncation renders binary search in b-tree nodes rather awkward but a variant of prefix truncation, offset-value coding, enables very efficient sequential search, possibly even aided by hardware (SIMD instructions).

Offset-value coding combines the size of the truncated prefix with the column value or the bytes following that prefix. Like prefix truncation, it can be applied to a string of characters, a string of bytes, or a string of columns, i.e., a database row. Offset-value coding speeds up row comparisons in a merge sort, both run generation and merging if both use tournament trees (also known as tree-of-losers priority queues). Special hardware instructions encapsulate the core logic of offset-value coding and tournament trees [20], [46].

While prefix truncation has been used in b-trees in the past [8], b-trees benefit from offset-value coding in sorting (e.g., during b-tree creation), in sorted intermediate query results (e.g., in b-tree scans), and in query execution (e.g., merge join). For example, after b-tree

scans translate prefix truncation into offset-value codes, a merge join can intersect lists of b-tree entries with hardly any comparisons beyond the offset-value codes. For another example, in a query such as "count (distinct B) group by A," the duplicate removal operation can sort on A,B, the grouping operation can count rows by A, and offset-value codes can aid the sort logic, the duplicate removal logic (offsets equal to key size indicate duplicate keys), and the grouping logic (offsets smaller than the grouping key indicate grouping boundaries). More details on offset-value coding in sorting and query execution can be found elsewhere [18], [33].

In summary of Section 5.5:

- Run-length encoding in column stores, sharing in trie data structures, prefix truncation in row stores, and offset-value coding all achieve the same compression. Offset-value coding combines the size of the truncated prefix with the column value or the bytes following that prefix.

- Offset-value coding speeds up sorting, b-tree creation, and query execution algorithms using their inputs' sort order, e.g., in-stream grouping, rollup, merge join, merge intersection, duplicate removal, and more. Advantages materialize in all cases of interesting orderings [68].

5.6 Summary of Query Processing

In summary of query processing with b-trees, new techniques continue to emerge. One wonders whether machine learning can improve not only an index structure but also usage patterns of b-tree indexes. In the meantime, in-memory b-tree indexes permit converging on a single algorithm for duplicate removal, grouping, and aggregation; perhaps the same is true for join, intersection, and other binary query operations. B-tree indexes and their insertion-optimized variants such as continuous merging with staggered key ranges also promise new opportunities in data warehousing and continuous information capture with instant analysis.

6

Concurrency Control

For transactional concurrency control, a few recent techniques may improve the behavior of b-trees in transaction processing. They are designed to avoid false conflicts by optimizing lock scopes (sizes, coverage – Section 6.1), lock semantics in multi-version storage (Section 6.2), lock durations (acquisition and release – Section 6.3), and the lock acquisition sequence (Section 6.4). Section 6.5 sketches alternative locking techniques, none truly satisfactory in every way, for concurrent insertions, reorganization, and queries in insertion-optimized b-trees.

A promising design is efficient compilation of scan predicates to machine code. The idea applies not only to scan, sort, and join operations but also to scan predicates for precision locks [48]. In optimistic concurrency control, a scan predicate can replace many row identifiers or key values in a transaction's read set [60]; in pessimistic concurrency control, a scan predicate can replace many shared locks. The scan predicate is tested against each row in another transaction's write set or each row with an exclusive lock. Note that there is no need to check two transactions' read sets or shared locks, because two read-only accesses are always compatible. Note also the similarity to end-of-transaction validation of predicates in IMS/VS Fast Path [22].

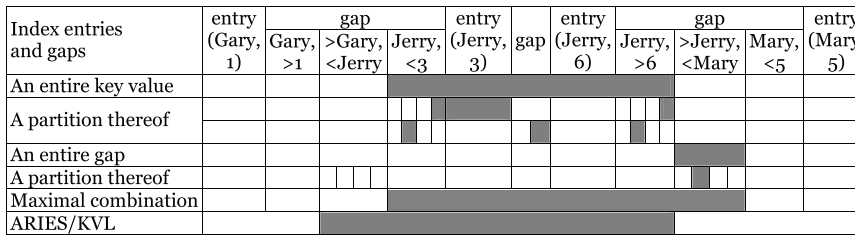

Figure 6.1: Lock scopes in orthogonal key-value locking.

6.1 Lock Scopes

In orthogonal key-value locking [32], a single lock request may cover (i) a key value with its entire set (in a non-unique index) of (existing and possible) rows (in a primary index) or of row identifiers (in a secondary index), (ii) a distinct key value and a subset of its (existing and possible) rows or row identifiers, (iii) a gap (open interval) between two (existing) distinct key values, (iv) a subset of (non-existing) key values within such a gap, or (v) any combinations of those.

Figure 6.1 illustrates orthogonal key-value locking for distinct key value "Jerry" in a non-unique secondary index on first names. The column headings indicate points and ranges in the domain of index keys. In addition to the lock scopes shown, orthogonal key-value locking also supports any combination. For example, a range query can, with a single invocation of the lock manager, lock a key value and its adjacent gap. This lock covers all existing and non-existing instances of the locked key value as well as all partitions of non-existing key values in the adjacent gap. For comparison, the last line of Figure 6.1 shows the one and only lock scope possible in ARIES/KVL [58], i.e., a distinct key value with all instances and an adjacent gap.

Figure 6.2 compares lock scopes in earlier methods and in orthogonal key-value locking. More specifically, it shows required and actual lock scopes for an unsuccessful query, i.e., for phantom protection for non-existing key value "Harry." Shading in the header indicates the gap that needs a lock. An S in Figure 6.2 indicates that a serializable locking technique acquires a transaction-duration shared lock in order to prevent

Index entries and gaps	entry (Gary, 1)	gap			entry (Jerry, 3)	gap	entry (Jerry, 6)	gap	
		Gary, >1	>Gary, <Jerry	Jerry, <3				Jerry, >6	>Jerry
ARIES/KVL			S on "Jerry"						
ARIES/IM		S on "3"							
KRL		S on "(Jerry, 3)"							
Orth. krl		NS on "(Gary, 1)"							
Orth. kvl			S						
w/ gap part'g									

Figure 6.2: Required and actual lock scopes in phantom protection for "Harry."

insertion of index value Harry. It is clear that ARIES/KVL locks the largest scope. ARIES/IM shows the same range as key-range locking (KRL) only because Figure 6.2 does not show the lock scope in other indexes of the same table. Orthogonal key-range locking locks less than key-range locking due to separate lock modes for index entry and gap. Lock mode "NS" (pronounced "key free, gap shared") indicates a shared lock on the gap between index entries but no lock on any index entry. Orthogonal key-value locking locks the smallest scope, in particular if the gap and its non-existing key values are partitioned as shown in the bottom row.

In summary of Section 6.1:

- The granularity of concurrency control is much more important for high concurrency than the choice between optimistic and pessimistic concurrency control.
- Among techniques for record-level locking in b-tree indexes and index-organized tables, orthogonal key-value locking stands out for separating existing key values from gaps between existing key values and for partitioning both the instances of existing key values and the non-existing key values in gaps.

6.2 Locking in Multi-Version Storage

For some reason, sometimes two-phase locking is associated with single-version storage and optimistic concurrency control is associated with multi-version storage. To be sure, optimistic concurrency control requires transaction-private update buffers or multi-version storage, whereas

locking does not. Nonetheless, locking can benefit from multi-version storage just as much as optimistic concurrency control.

More specifically, if all transactions acquire locks and release them at end-of-transaction, i.e., all transactions execute with their commit point at end-of-transaction, two versions per data item are sufficient. One version is for all readers and the other one is uncommitted, locked by a writer. On the other hand, if read-only transactions run in snapshot isolation with their commit points at start-of-transaction, then multiple versions per data items are desirable. Old versions become obsolete and subject to garbage collection when their replacement (successor version) has been committed before the oldest active transaction started.

Locks and their semantics are somewhat different in multi-version storage than in single-version storage. As indicated above, shared locks for a logical data item protect the most recent committed version, whereas an exclusive lock protects the one-and-only uncommitted version. An updating transaction must not commit (and thus modify what is the most recent committed version) while another transaction holds a shared lock [6]. Something stronger than a standard X locks is required during commit processing.

Figure 6.3 summarizes the rules for choosing versions. Record-level locking, e.g., key-range locking in b-trees, ignores versions and instead focuses on index entries. One transaction's shared lock does not prevent another transaction from creating of a new, uncommitted version but it does prevent commit of a new version. A read-only transaction in snapshot isolation reads the version appropriate for its start-of-transaction timestamp, a version creator reads its own updates, and all other transactions read the most recent committed version. All new locking techniques must work with multi-version storage in this way.

Transaction mode	Lock mode	Version access
Read-only	Snapshot isolation, no locks	Read the version appropriate for the start-of-transaction timestamp
Read-write	Shared lock	Read the most recent committed version
	Exclusive lock	Create, modify, and read an uncommitted version

Figure 6.3: Transactional data access in versioned databases.

In summary of Section 6.2:

- For clean transaction semantics, snapshot isolation is best limited to read-only transactions and best supported by multi-version records.
- For read-write transactions, shared locks give access to the most recently committed version and exclusive locks permit creating, editing, and reading a new uncommitted version.
- Committing a new version changes the most recently committed version and should be permitted only with no active readers (i.e., no conflicting shared locks).

6.3 Lock Durations

In traditional implementations of transactions and their concurrency control, lock acquisition occurs before the first access to a data item in the database and lock release occurs after a transaction is hardened, i.e., when the transaction's commit log record is safely in the recovery log on stable storage. Two more techniques, in addition to orthogonal key-value locking and multi-version storage, reduce the probability of lock conflicts and deadlocks. The older one of these, controlled lock violation, weakens exclusive locks during hardening; the other technique weakens locks during a transaction's initial "read phase," i.e., while it executes its application logic, accesses database contents, and acquires locks.

Controlled lock violation [40] weakens locks while a transaction is hardening, i.e., while a transaction waits for its commit log record to reach stable storage. Once a transaction's commit log record has a place in the recovery log and thus a log sequence number, its position in

the equivalent serial history and thus in the database history is fixed. The need for concurrency control ceases, except that no transaction may commit that depends on an update not yet durably committed. In controlled lock violation, another transaction may read and even over-write a recent update, but it must not commit until the recent update is durable on stable storage. Thus, the cost of lock violation is a commit dependency. If the dependent transaction is a read-write transaction, this is negligible; if it is read-only, e.g., a local read-only participant in a distributed read-write transaction, the local participant delays its response during two-phase commit until the earlier update transaction is durable.

With deferred lock enforcement [31], a read-write transaction in its read phase acquires and holds exclusive locks, but only as weak locks. More specifically, deferred lock enforcement interprets exclusive locks as reserved locks during a transaction's read phase, as pending locks while a transaction is preparing for commit, and as traditional exclusive locks only while a transaction is committing. Importantly, the switch from reserved to pending is accomplished with a simple change in the transaction's state in the transaction manager. All pending locks drain conflicting reader transactions concurrently. Merely the verification that all conflicting reader transactions have indeed released their locks is serial.

<table>
<tr><td rowspan="2">Transaction phases →
↓ Techniques</td><td rowspan="2">Read phase
= transaction & application logic
10^5 inst = 20 µs</td><td colspan="3">Commit logic</td><td rowspan="2">Hardening
= force log to stable storage
1 I/O = 200 µs</td></tr>
<tr><td>Commit preparation</td><td>Commit log record
10^3 inst = 0.2 µs</td><td>Update propagation</td></tr>
<tr><td>Traditional locking</td><td></td><td rowspan="2">n/a</td><td></td><td rowspan="2">n/a</td><td></td></tr>
<tr><td>Controlled lock violation</td><td></td><td></td><td></td></tr>
<tr><td>Deferred lock enforcement</td><td></td><td colspan="3"></td></tr>
<tr><td>Both dle and clv</td><td></td><td colspan="2"></td><td></td></tr>
<tr><td>Dle, clv, multi-version storage</td><td></td><td colspan="2"></td><td>n/a</td><td></td></tr>
</table>

Figure 6.4: Lock enforcement durations.

Figure 6.4 compares techniques from traditional locking to the recommended combination of deferred lock enforcement, controlled lock violation, and multi-version storage. With read-only transaction in snapshot isolation, this diagram is only about read-write transactions. Long-running transaction phases are shaded in the header row. Dark shading indicates traditional, strict lock enforcement and light shading indicates weak locks, i.e., update locks during the read phase and a commit dependency after lock violation during hardening. Traditional locking acquires and holds all locks during a transaction's two long-running phases, which indubitably has contributed to giving locking and serializability a reputation for poor concurrency, poor scalability, and poor system performance. In contrast, the combination of three techniques avoids strict locks during both long-running phases and one step in commit processing. More specifically, controlled lock violation avoids strict locks during hardening, early detection of write-write conflicts and multi-version storage render transaction-private update buffers and update propagation unnecessary, and deferred lock enforcement avoids all conflicts (other than write-write conflicts) among transactions executing their application logic during their read phase. Therefore, the combined techniques reduce the effective duration of exclusive locks to the bare minimum: concurrent waiting for truly exclusive access followed by formatting a commit log record.

In summary of Section 6.3:

- Weak locks as defined in deferred lock enforcement and controlled lock violation avoid many false conflicts and permit more concurrency than any other variant of optimistic or pessimistic concurrency control, yet ensure perfect serializability.

- Controlled lock violation, deferred lock enforcement, and orthogonal key-value locking minimize false conflicts and ensure perfect serializability with maximum concurrency.

6.4 Lock Acquisition Sequences

The frequency of deadlocks depends on the lock acquisition sequences in queries and updates. Query execution plans in relational databases often

search a secondary index and then fetch data from a table's primary storage structure. In contrast, database updates usually modify first a table's primary storage structure and then all affected secondary indexes. Deadlocks are likely if queries acquire locks first in the secondary index and then in the primary storage structure while updates acquire locks first in the primary storage structure and then in secondary indexes. Using the traditional, opposing navigation in queries and updates but the same locking sequence in both queries and updates retains the efficiency of both queries and updates but avoids deadlocks caused by the traditional, opposing lock acquisition sequences [30].

The proposed new lock acquisition sequence pertains to updates and to index maintenance plans. Before modifying a table's primary storage structure, in fact before lock acquisition within the table's primary storage structure, the new lock acquisition sequence acquires all required locks in all affected secondary indexes, with orthogonal key-value locking recommended. In this design, the update and maintenance operations that actually modify the secondary indexes do not acquire any further locks, e.g., key-range locks or key-value locks.

The sequence of database accesses and of database updates remains unchanged – only the lock acquisition sequence changes. Thus, buffer pool management, log record creation, etc. all remain unchanged. Even lock release remains unchanged: early lock release [17] or controlled lock violation [40] remain possible and, in fact, recommended.

Key-range locking and key-value locking must not acquire locks on index keys that do not exist. Thus, in most implementations, a thread retains a latch on an index leaf page while requesting a lock on a key value. (Waiting and queuing require special logic for latch release.) In the new regimen, it is not possible to hold a latch on an index page during lock acquisition for index keys. Thus, it might appear at first as if the new technique could attempt to lock non-existing key values. Fortunately, this is not the case. It is sufficient to hold a latch on the appropriate data page in the table's primary storage structure during acquisition of locks in secondary indexes. This latch guarantees that key values in secondary indexes remain valid during lock acquisition.

Deletions present hardly a problem – the deletion logic while accessing the table's primary storage structure can readily obtain all column

values required for the secondary indexes and request the correct locks. This includes a bookmark if it is part of a unique entry in a secondary index and thus part of a lock identifier.

The original descriptions of ARIES locking assumed immediate removal of index entries, which required locks on a neighboring key in addition to the key being deleted [58], [59]. The update of a table's primary storage structure cannot anticipate neighboring key values in all affected secondary indexes. Thus, their deletion logic does not permit lock acquisition while modifying the table's primary storage structure, i.e., before accessing the appropriate leaf page in all secondary indexes. Later ARIES work recommends logical deletion, i.e., turning a valid record into a pseudo-deleted record and relying on subsequent asynchronous clean-up, e.g., as part of a later insertion attempt. Toggling the ghost (pseudo-deleted) bit in an index entry requires a lock only on the index entry or its key value but not its neighbor. Thus, deletion via ghost status, now the standard way of implementing deletion in database systems, enables the change in the sequence of lock acquisitions.

Insertions present a different problem: the required key values might not exist yet in the affected secondary indexes. In general, locks on non-existing key values seem like a bad idea. For example, Tandem's solution to phantom protection inserts a key value into the lock manager but not the index leaf; anecdotal evidence suggests long and onerous testing and debugging. Here, however, the locked key value will soon exist in the secondary index, created by the insertion transaction acquiring the lock. As soon as a system transaction creates the required new ghost index entry, invoked by the insertion for the secondary index, the system reaches a standard state and the user transaction can turn the locked ghost into a valid index entry.

Updates of non-key index fields, i.e., those that do not affect the placement of the entry within the index, permit the new timing of lock acquisition in secondary indexes. There is no issue with non-existing key values. Updates of key fields require deletion and insertion operations and should be locked as described above.

In summary of Section 6.4:

- With queries locking first secondary indexes and then the primary storage structure, and with updates locking first the primary storage structure and then secondary indexes, most deadlocks are between queries and updates.
- With updates also locking secondary indexes first, these (i.e., most) deadlocks are eliminated.

6.5 Concurrency Control Within Insertion-Optimized B-Trees

If all operations on an insertion-optimized b-tree are queries in snapshot isolation and blind insertions of new information, b-trees (within a forest of b-trees) or partitions (within a partitioned b-tree) represent a suitable granularity of concurrency control, e.g., of locking. If, however, the database, table, or index must also support traditional (and serializable) read-write transactions including reads that must logically occur at end-of-transaction, i.e., at the commit point of transactional writes, then a finer granularity of locking is required.

The principal difficulty is that key values required in a read-write transaction might be located in any recent partition. Thus, a direct application of traditional key-range locking or key-value locking requires setting locks in any recent partition. Such overhead is clearly undesirable.

Instead, traditional key-range locking or key-value locking may focus on actual key values in the largest and most stable partition. An advantage is that a traditional lock manager and its hash table suffice. One issue is that lock acquisition must find and retrieve the key value to be locked, thus incurring I/O and polluting the buffer pool. Another issue is that occasional reorganization and merge steps modify the sets of key values in the largest partition, and that this requires adjustment of any locks held during that time.

Alternatively, traditional key-range locking or key-value locking may focus on the in-memory partition. This avoids one issue with the prior design, but it severely aggravates the other issue, because spilling and reorganization of the in-memory partition is not occasional but continuous.

A design used in at least one real system (Spanner [4], [13]) separates locks and lock scopes from the actual key values in any partition. Instead, a query predicate defines a lock and its scope or key range, and the lock manager indexes, searches, and intersects intervals (key ranges). This is quite different from a traditional lock manager and its lookup mechanism. Instead of a hash table, it requires an in-memory index for intervals.

In summary of Section 6.5:

- As much as orthogonal key-value locking avoids false conflicts with few lock manager invocations, it is merely one of several imperfect solutions to locking in partitioned b-trees and in log-structured merge-forests.
- Inasmuch as the in-memory b-tree caches not only recent insertions but the application's entire working set, orthogonal key-value locking in the in-memory b-tree partition is a good solution for concurrency control for queries and transactions in log-structured merge-forests.

6.6 Summary of Concurrency Control

In summary of concurrency control for b-tree indexes, there are some techniques omitted in the earlier survey [29] as well as some techniques invented since then. The former group includes locking rules for multi-version index entries; the latter include orthogonal key-value locking in b-trees, lock acquisition sequences for deadlock avoidance in primary and secondary indexes, as well as deferred lock enforcement and controlled lock violation in any database transaction.

7

In-Memory B-Trees

While b-trees were invented for persistent indexes on disk storage, they can be useful in any step in the memory hierarchy with block transfers. For example, b-tree nodes equal to cache lines have been proposed for in-memory indexes optimized for efficient use of CPU caches, as well as b-trees optimized for both cache lines and disk pages [10]. Other example steps in a memory hierarchy occur between storage in flash memory and rotating magnetic disks, local and remote memory, high-bandwidth memory (HBM) and traditional DRAM memory, etc. In other words, b-trees can be superior to binary trees even in memory and even in the absence of persistent or page-oriented external storage devices.

7.1 Applications of In-Memory B-Trees

B-trees are versatile in memory just as much as on persistent storage devices. They are superior to hash tables in particular if ordering or range predicates matter. They are also superior if serializable transactions matter, e.g., phantom protection by locking the absence of key values in the range between two existing key values. The tree structure

of b-trees is also required for hash tables if very large continuous arrays (address ranges) are difficult or expensive; and the direct address calculation of hash tables can be emulated by interpolation search in b-tree nodes [27], [65], with a linear regression calculation perhaps labeled "learned" indexes [50]. A promising hybrid is a b-tree on hash values, with phantom protection based on key ranges of hash values and with direct address calculation based on interpolation search among uniformly distributed hash values; efficient extraction of pseudo-random yet repeatable samples is an added benefit of b-trees on hash values.

B-trees in memory can be thread-private, e.g., for immediate in-memory aggregation during the input phase of an external merge sort supporting duplicate removal, grouping, aggregation, and pivoting [19]. B-trees in memory can also be shared among threads, e.g., in a client-side cache of a server-side database or in an in-process database library. Moreover, b-trees in memory can support transactions with ACID properties (all-or-nothing failure atomicity, consistency, multi-user isolation or concurrency control, and "come fire, flood, or insurrection" durability [Lindsay]), e.g., as part of an in-memory database server. Finally, in-memory b-trees can support key value storage (get-put-commit), databases (select-from-where), file systems (open-read-close), or caches for any of these.

More than the usage or application, what distinguishes the implementations of in-memory b-trees is their support for threading, transactions, recovery, and availability. After brief reviews of data structures and algorithms for in-memory b-trees, the following sub-sections focus on implementation techniques for these distinguishing characteristics.

In summary of Section 7.1:

- B-trees in memory are more efficient than hash tables with respect to creation (sorting), serializability (phantom protection), range predicates, and merge join.
- B-trees on uniformly distributed keys (e.g., hash values) permit effective interpolation search.

7.2 B-Tree Structure in Memory

In principle, the format and structure of b-trees in memory is the same as of b-trees on disk or other external persistent storage and, after all, in an in-memory buffer pool. Obviously, the format of pointers is different: in traditional b-trees, a parent-to-child pointer or a sibling pointer (if those exist) is a page identifier, e.g., a page number within a database file; whereas in an in-memory b-tree, it is an in-memory address, typically an address in virtual memory.

But even this difference is questionable: after a detailed profiling effort found that a substantial fraction of CPU cycles in a transactional database system was spent in the buffer pool [45], pointer swizzling was found to eliminate most of this effort if the application's working set fits in the buffer pool, i.e., with limited turn-over in the buffer pool [42]. Pointer swizzling here means replacing or augmenting node-to-node pointers, e.g., parent-to-child pointers, with in-memory addresses while both pointer source and target remain in the buffer pool. Obviously, page replacement in the buffer pool requires un-swizzling. Node splits require careful pointer adjustments, in particular splits of branch nodes. Reverse pointers, e.g., child-to-parent pointers, simplify the required logic. They are practical if used only for swizzled pointers. Interestingly, pointer swizzling in a buffer pool renders all b-trees in-memory b-trees, even if implementations differ in their support for multiple threads, multiple transactions, concurrency control, and durability.

Variable-size b-tree entries, e.g., strings or fixed-size keys and values after compression, require either in-node pointers or memory management outside of the b-tree and its nodes. For in-node pointers, 2-byte offsets (byte counters) are often sufficient and more space-efficient than 8-byte virtual-memory addresses. Out-of-node strings and memory management seem counter-intuitive to the idea of b-trees, which were invented specifically to map storage, search, and updates of variable-size collections to simple collections of fixed-size containers.

With memory space always expensive due to machine limitations, purchasing cost, and requirements for power and cooling, effective and efficient memory management including compression is needed for in-memory b-trees perhaps even more than for b-trees on external storage.

Simple forms of compression are prefix- and suffix-truncation for keys [5] as well as all traditional compression technique for values associated with keys. For keys, order-preserving compression is required, e.g., arithmetic coding, order-preserving Huffman codes, or order-preserving dictionary encoding [73]. CPU caches and the relatively high cost of cache faults render compression perhaps even more important, as well as other optimizations specifically for cache efficiency. For example, variable-size b-tree entries are usually managed with an array of offset-length pairs next to a node header; for cache efficient search, pairs of offset and key prefix might be better.

There are also designs for b-trees within b-tree nodes, e.g., O'Neil's SB-trees [62]. In a multi-level memory hierarchy, e.g., memory and flash storage and disk drives, with different access latency and transfer bandwidth and therefore different page sizes, a b-tree within each b-tree node suggests itself, e.g., a b-tree of flash pages (e.g., 1–4 KB) within a b-tree of disk-array pages (e.g., $\frac{1}{8}$–$\frac{1}{2}$ MB). A possibly significant concern is that a space utilization of 70% at each of two levels means overall space utilization of less than 50% [$0.7 \times 0.7 = 0.49$]. On the other hand, 50% space utilization is considered a standard price to pay for reliability and availability, e.g., in the form of mirroring or replication. But then again, mirroring such a two-level b-tree means overall space utilization of less than 25% [$0.49 \times 0.5 = 0.245$].

An entirely different approach to efficient search within b-tree nodes focuses on simple in-node data structures and algorithms, e.g., sequential search. There are multiple ways to speed up sequential search. First, the search should focus on fixed-size fixed-type pieces of the keys – poor man's normalized keys are the obvious choice, e.g., a prefix of the normalized key cast to an unsigned integer. Second, the array to search can be dense − offsets or pointers to full index entries should be relegated to a parallel array. Third, instead of poor man's normalized keys extracted from each key value after page-wide prefix truncation, the search may use offset-value coding [12], i.e., encoding each key value relative to its immediate predecessor within the b-tree's sort order. In this case, the search for one or multiple key values within a sorted array is more akin to a merge than a search. Fourth, widely available SIMD

instructions can provide limited hardware support for search within an array of offset-value codes.

In addition to "search as merge" sped up by offset-value codes, possibly with hardware support, another important optimization for in-memory b-trees focuses on insertions and the memory hierarchy, e.g., from CPU cache via high-bandwidth memory (HBM) and traditional DRAM memory to non-volatile ("storage-class") memory attached to the system bus or a fast I/O channel such as PCIe. In principle, each step in the memory- or storage-hierarchy deserves its own b-tree, page size, free space, etc. Staging (moving nodes up and down in the memory hierarchy), search (as discussed above) in the context of bulk insertions require thoughtful mechanisms and policies.

High-bandwidth insertions into an ordered index can employ a never-ending external merge sort: a small b-tree high in the memory hierarchy, b-trees in lower levels of the memory hierarchy organized by size and merged to limit their number and thus the effort required in each query. Small fast b-trees apply to a single bulk insertion or a rapid sequence of small insertion transactions, which in turn includes insertions of tombstone records, i.e., logical deletions, and of replacement records, i.e., logical updates. Traditional designs include log-structured merge-forests [63], stepped-merge trees [47], and their many recent variants [56]. Related designs include partitioned b-trees [25] and continuous merging [24], i.e., carefully-paced but never-ending merge steps with runs entering and exiting merge steps at staggered key values. All these designs share the idea of using a small b-tree in order to build a large b-tree. These designs and their merge policies create a continuum between traditional b-trees (fast low-latency queries but high write amplification for updates) and differential files [69] adapted to b-trees (high insertion bandwidth but poor query performance). Merging in-memory runs in order to optimize cache usage and cache transfers obviously applies not only to sorting [61] but also to b-trees and their maintenance.

In summary of Section 7.2:

- In-memory b-trees use memory addresses as pointers – but b-tree nodes in a buffer pool can also use "swizzled" pointers for efficient b-tree navigation.

- Compression can be valuable for b-trees in memory just as much as for b-trees in a buffer pool.
- Page-wide prefix truncation compresses less than key-to-key prefix truncation. The former permits binary search and interpolation search but the latter does not.
- A variant of prefix truncation, offset-value coding, enables very efficient sequential "search by merge," which may be sufficient or even superior within a b-tree node.
- High-bandwidth b-tree maintenance, in particular insertions (including replacement records and tombstone records), resembles external merge sort with initial runs, merging, etc.

7.3 Low-Level Concurrency Control

There are two levels of concurrency control in database software: low-level concurrency control coordinates threads in order to protect in-memory physical data structures whereas high-level concurrency control coordinates transactions in order to protect logical contents of databases or indexes [71]. Low-level concurrency control is the realm of spinlocks, latches, critical sections, hardware-transactional memory and lock-free data structures; high-level concurrency control is the realm of serializability theory, phantom protection, two-phase commit, hierarchical locking, and more.

B-trees require low-level concurrency control for contents-neutral operations, i.e., structure modifications that do not modify the logical contents of the b-tree index. Typical examples include splitting an overflowing b-tree node in two (or splitting two nodes into three), load balancing among neighboring nodes, and node removal. In addition to these traditional structure modifications, transactional b-trees often use ghost records (also known as invalid or pseudo-deleted records) to split a deletion into two steps: the user transaction marks a record invalid by flipping the record's ghost bit, and a system transaction erases the ghost record after the user transaction commits and cannot possibly require a rollback. System transactions perform all allocation actions

including the node operations above as well as creation and removal of ghost records. As they are contents-neutral, system transactions require only low-level concurrency control. They also have special semantics with respect to atomicity and durability, e.g., in write-ahead logging and logging.

In-memory b-trees also benefit from contents-neutral system transactions. Their low-level concurrency control can be provided with traditional read-write latches, with read-only shared latches, or with lock-free operations. Read-only shared latches do not count concurrent readers, i.e., readers do no modify the data structure representing the latch. Instead, writers increment a version counter within the latch data structure upon entry and exit of the critical section. Readers entering a critical section check for an even version counter and retain its current value; when exiting, they check the version counter has remained unchanged. In other words, both latch acquisition and release can fail the critical section.

Lock-free create shadow copies of nodes in a tree-shaped data structure and, using a single atomic test-and-set instruction, switch from the old tree contents to the new. In addition to the limit to tree shapes, lock-free data structures suffer from the lack of a shared or read-only mode: the remedy is to omit anything equivalent to shared latches or to apply the equivalent of exclusive latches, i.e., create new versions even for read-only parts of the tree shape.

For in-memory b-trees, all three mechanisms for low-level concurrency control are viable options: traditional latches, read-only shared latches, or lock-free thread coordination. The best choice depends on the multi-programming level, on the read-write ratio, and on contention among critical sections. Each of these three choices has its advantages and disadvantages.

In summary of Section 7.3:

- Low-level concurrency control coordinates threads in critical sections (as opposed to high-level concurrency control coordinating transactions) – low-level concurrency control is required to coordinate contents-neutral structural b-tree changes, e.g., node splits.

- Lock-free low-level concurrency control suffers from the lack of a shared or read-only mode, even in a hierarchical data structure such as a b-tree.

7.4 High-Level Concurrency Control

Transaction coordination to protect logical contents is, in a first approximation, the same for in-memory databases and indexes as for traditional databases and indexes on external storage devices. Thus, all techniques of Section 6 apply to in-memory b-trees, including orthogonal key-value locking, lock semantics for multi-version indexes, deferred lock enforcement, controlled lock violation, and aligned lock acquisition sequences for queries and updates.

Optimistic concurrency control is often considered for in-memory databases, indexes, and b-trees. There are three forms of end-of-transaction conflict detection. Instead of the original design [51], later named backward validation, or the alternative forward validation [44], end-of-transaction timestamp validation is a frequent proposal. If phantom protection is desired, a gap between adjacent key values or index entries can be guarded by an adjacent key value or index entry, as in key-value locking and key-range locking [32], [54], [58]. Note that insertion into a gap must modify the timestamp of the key value or index entry that guards the gap. Note also that the original design and traditional serializability semantics require atomicity for each transaction's validation and write phases together [51], not just for each validation phase.

Distributed transactions under optimistic concurrency control require two-phase commit. In the first phase, each participant voting to commit firmly pledges to implement the coordinator's global commit decision. This pledge must not be subject to subsequent end-of-transaction validation of the participant transaction or any other local transaction. Inasmuch as locks are the best way to implement such a pledge, optimistic concurrency control is a form of deferred lock acquisition [31]. In fact, optimistic concurrency control with concurrent validation of multiple transactions requires coordination among validating transactions and similarly is a form of deferred lock acquisition. Note that

this interpretation of end-of-transaction validation immediately permits controlled lock violation [40] during a transaction's write phase, while hardening to the recovery log, or while waiting for the coordinator's global commit decision.

In-memory databases and their b-trees, due to their lack of I/O delays, have given rise to deterministic transaction scheduling, where groups of transactions are executed without concurrency control after a prior check for conflicts. One could be tempted to describe this approach as "start-of-transaction validation," i.e., just the opposite of optimistic concurrency control. One could also be tempted to describe it as "log shipping to self," because it first constructs a schedule of the work and then, much like the recipient in traditional log shipping, executes this schedule without concurrency control. It seems immaterial whether this log is logical ("command logging"), physio-logical [43] (logical updates within specific database pages), or physical (byte-for-byte change descriptions).

An entirely different take on in-memory b-trees focuses on implementation techniques for a lock manager, whether the locked objects reside in memory or on external storage. Locks represent a many-to-many relationship between transactions and database objects (or their hash values). As usual for many-to-many relationships, there are two principal access patterns: navigation from transactions to locks (and database objects), e.g., during lock release at end-of-transaction; and navigation from database objects to locks (and transactions), e.g., checking for lock conflicts during lock acquisition. The standard representation of locking information uses a double-linked list of locks per transaction plus another double-linked list of locks per database object (or per hash bucket). Low-level concurrency control (latching, lock-free maintenance) for this web of pointers is rather difficult. On the other hand, another standard representation of many-to-many relationships is a pair of indexes in support of bi-directional navigation. Thus, the question arises whether a pair of in-memory b-trees (or two partitions within a single in-memory b-tree), of course with suitable low-level concurrency control, can represent lock information with efficient bi-directional navigation and efficient maintenance. Would 1,000 CPU cycles per b-tree update suffice for lock manager performance; and are 1,000 CPU cycles

per update achievable in a b-tree of moderate size? Could hierarchical locking such as orthogonal key-value locking [32] reduce the frequency of lock acquisitions, of lock manager invocations, and of lock table maintenance?

In summary of Section 7.4:

- High-level concurrency control coordinates transactions in order to protect logical database and b-tree contents (not their representation).
- Key-range locking, key-value locking, etc. apply to in-memory transactions and in-memory storage structures in exactly the same ways as to databases and b-trees on external storage.
- Optimistic concurrency control is popular for in-memory databases and b-trees but distributed transactions require two-phase commit and thus firm promises and locks – optimistic concurrency control can be seen as a form of deferred lock acquisition, in particular in the contexts of concurrent validation and of two-phase commit.

7.5 Failures and Recovery

Errors and failures seem inevitable, both in hardware and in software. B-trees are no exception. One possible approach to the problem relies on self-repairing b-trees [39], which combine continuous consistency checking and on-demand "instant" single-page repair. The central ideas are consistency checking during normal b-tree traversals and write-ahead logging with log records linked by database page. Continuous and comprehensive consistency checking is not possible for some forms of b-trees due to the "cousin problem," i.e., constraints on pointers and key ranges between neighboring nodes that share a grandparent (or further ancestor) node but not a parent node. Avoiding neighbor pointers and adding fence keys to b-tree nodes permits comprehensive consistency checking during normal b-tree traversals. Foster b-trees [35] combine continuous and comprehensive consistency checking, local overflow nodes for efficient structure modifications without low-level concurrency control against the usual top-down b-tree traversals, and a

rigorous separation of updates to logical contents by user transactions and updates to the b-tree structure by system transactions.

Instant single-page repair opens many opportunities for incremental on-demand recovery. For example, instant restart after a system failure (software crash) permits new transactions and new checkpoints immediately after log analysis; and instant restore after a media failure (storage hardware) permits new transactions immediately if the server state (transaction manager, lock manager, buffer pool manager) survive the failure. For in-memory databases and their b-trees, instant reboot combines the techniques of instant restart and instant restore, in effect applying the logic for a double failure in a traditional database with external storage. Note that the logic and advantages of single-phase restore [66] also apply to in-memory databases and b-trees, i.e., merging backup(s) and (sorted) log archive and thus avoiding a separate recovery phase for log replay. Note also that the logic for instant reboot enables instant fail-over to a cold (out-of-date) replica, whether the data is in memory or on external storage devices.

B-trees on persistent memory promise high performance and fast recovery but introduce a separate problem. Write-ahead logging requires that log records arrive on stable storage before in-place updates in the database. A possible remedy, sometimes called "log shipping to self," first formats log records and forces them to the recovery log on stable storage and then invokes their "redo" method to apply an in-place update. In fact, this approach promises to avoid some redundant code: without any changes to "redo" and "undo" methods, "do" methods merely create log records but have no code of their own to modify the data store.

Finally, in-memory b-trees may be used in a "log shipping to storage" design pioneered in Aurora [70]. The primary data server manages no local storage and does not write anything when a dirty page is evicted from the buffer pool. It does, however, send all its log records to a storage server, which applies these log records to its copy of the database. When the primary data server incurs a buffer pool fault, the storage server serves up-to-date pages. Application of log records may be synchronous or asynchronous, and the latter may be eager or on-demand when an up-to-date page image is needed. The storage server may also sort and

index log records by the affected database page in order to enable instant repair as well as single-phase restore. In contrast to log shipping to storage, traditional network-attached storage and modern disaggregated storage require writing (i.e., sending) up-to-date database pages. Note that either design requires that the primary data server applies log records to page images in its local buffer pool.

In summary of Section 7.5:

- B-trees permit continuous comprehensive consistency checking, i.e., including in-page plausibility as well as structural constraints such as key ranges and pointers, if there is always only a single pointer per node, e.g., in foster b-trees.
- Single-page repair enables not only incremental on-demand "redo" and thus seemingly instant restart after a system failure but also self-repairing b-trees as well as instant reboot after loss of an in-memory database including all server state and storage contents.

7.6 Summary of In-Memory B-Trees

In summary, in-memory b-trees can be thread-private or shared (which requires concurrency control); and transient (lost in a software crash) or permanent (with a persistent copy or recovery log). In many other respects, they are like b-trees on external storage devices – they can be local or distributed; and single-copy or replicated (mirrored). They require low-level concurrency control for multi-threading and high-level concurrency control for transactional queries and updates; and they require logging or mirroring for recovery from failures and for high availability. With an in-memory buffer pool that supports pointer swizzling, the in-memory working set of externally stored b-trees mostly behaves like an in-memory b-tree.

8

Summary and Conclusions

In summary, many improvements to b-tree structure, contents, and usage have been invented since publication of an earlier survey [29]. With those, b-trees might well strengthen their role as universal data structure for reliable and efficient data management. For example, if business intelligence and machine learning require efficient scans of few columns but many rows within a table, column storage is superior to traditional database storage formats; but with b-trees suitably formatted and compressed, they emulate column storage and replicate its advantages very well. In fact, it could be argued that

- b-trees can improve upon traditional zone maps by exploiting their hierarchical structure,
- Partitioned b-trees match the epochs etc. of read- and write-optimized column storage,
- Continuous and adaptive merging can readily be implemented using scan, search, merge, concurrency control, logging, and recovery techniques already available for b-trees,

- Linear partitioned b-trees and near-linear partitioned b-trees reduce the count of root-to-leaf traversals in query processing over traditional forests of b-trees.

It seems that b-tree technology continues to evolve even half a century after their introduction, perhaps for decades to come.

Acknowledgements

Acknowledgements: Thanh Do, Wey Guy, and the reviewers suggested numerous valuable improvements in focus, scope, organization, and presentation of this work.

References

[1] S. Agarwal, J. A. Blakeley, T. Casey, K. Delaney, C. A. Galindo-Legaria, G. Graefe, M. Rys, and M. J. Zwilling, *Microsoft SQL server (Chapter 27),* in *Database System Concepts*, 4th edition. 2001, pp. 969–1006.

[2] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis, "Weaving relations for cache performance," *VLDB Conference*, 2001, pp. 169–180.

[3] G. Antoshenkov, "Dictionary-based order-preserving string compression," *The VLDB Journal*, vol. 6, no. 1, 1997, pp. 26–39.

[4] D. F. Bacon *et al.*, "Spanner: Becoming a SQL system," *ACM SIGMOD Conference*, 2017, pp. 331–343.

[5] R. Bayer and K. Unterauer, "Prefix b-trees," *ACM TODS*, vol. 2, no. 1, 1977, pp. 11–26.

[6] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[7] D. Bitton and D. J. DeWitt, "Duplicate record elimination in large data files," *ACM TODS*, vol. 8, no. 2, 1983, pp. 255–265.

[8] M. J. Carey, D. J. DeWitt, M. J. Franklin, N. E. Hall, M. L. McAuliffe, J. F. Naughton, D. T. Schuh, M. H. Solomon, C. K. Tan, O. G. Tsatalos, S. J. White, and M. J. Zwilling, "Shoring up persistent applications," *ACM SIGMOD Conference*, 1994, pp. 383–394.

[9] A. Chan, S. Fox, W.-T. K. Lin, A. Nori, and D. R. Ries, "The implementation of an integrated concurrency control and recovery scheme," *ACM SIGMOD Conference*, 1982, pp. 184–191.

[10] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin, "Fractal prefetching b-trees: Optimizing both cache and disk performance," *ACM SIGMOD Conference*, 2002, pp. 157–168.

[11] D. Comer, "The ubiquitous b-tree," *ACM Computing Surveys*, vol. 11, no. 2, 1979, pp. 121–137.

[12] W. M. Conner, "Offset value coding," *IBM Technical Disclosure Bulletin*, vol. 20, no. 7, 1977, pp. 2832–2837.

[13] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems*, vol. 31, no. 3, 2013, 8:1–8:22.

[14] N. Dayan, M. Athanassoulis, and S. Idreos, "Monkey: Optimal navigable key-value store," *ACM SIGMOD Conference*, 2017, pp. 79–94.

[15] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *OSDI*, 2004, pp. 137–150.

[16] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *CACM*, vol. 53, no. 1, 2010, pp. 72–77.

[17] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. Stonebraker, and D. A. Wood, "Implementation techniques for main memory database systems," *ACM SIGMOD*, 1984, pp. 1–8.

[18] T. Do and G. Graefe, "Robust and efficient sorting with offset-value coding," *ACM TODS*, vol. 48, no. 1, 2023, 2:1–2:23.

[19] T. Do, G. Graefe, and J. Naughton, "Efficient sorting, duplicate removal, grouping, and aggregation," *ACM TODS*, vol. 47, no. 4, 2022, 16:1–16:35.

[20] Enterprise system architecture/370, principles of operation. IBM publication SA22-7200-0, August 1988. CFC "compare and form codeword" and UPT "update tree" instructions.

[21] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, 1960, pp. 490–499.

[22] D. Gawlick and D. Kinkade, "Varieties of concurrency control in IMS/VS fast path," *IEEE Data Engineering Bulletin*, vol. 8, no. 2, 1985, pp. 3–10.

[23] N. Glombiewski, B. Seeger, and G. Graefe, "Waves of misery after index creation," *BTW Conference*, 2019, pp. 77–96.

[24] N. Glombiewski, B. Seeger, and G. Graefe, "Continuous merging: Avoiding waves of misery in tiered log-structured merge trees, unpublished," 2021.

[25] G. Graefe, *Sorting and Indexing with Partitioned B-Trees*. CIDR, 2003.

[26] G. Graefe, "Write-optimized b-trees," *VLDB Conference*, 2004, pp. 672–683.

[27] G. Graefe, "B-tree indexes, interpolation search, and skew," *DaMoN*, 2006, p. 5.

[28] G. Graefe, "Efficient columnar storage in b-trees," *SIGMOD Record*, vol. 36, no. 1, 2007, pp. 3–6.

[29] G. Graefe, "Modern b-tree techniques," *Foundations and Trends in Databases*, vol. 3, no. 4, 2011, pp. 203–402.

[30] G. Graefe, "Avoiding index-navigation deadlocks," in *Synthesis Lectures on Data Management*, Morgan & Claypool Publishers, 2019.

[31] G. Graefe, "Deferred lock enforcement," in *Synthesis Lectures on Data Management*, Morgan & Claypool Publishers, 2019.

[32] G. Graefe, "Orthogonal key-value locking," in *Synthesis Lectures on Data Management,* Morgan & Claypool Publishers, Extended from Goetz Graefe, Hideaki Kimura: Orthogonal Key-Value Locking, *BTW Conference*, pp. 237–256.

[33] G. Graefe and T. Do, "Offset-value coding in database query processing," *EDBT Conference*, 2023, pp. 464–470.

[34] G. Graefe, W. Guy, and C. Sauer, "Instant recovery with write-ahead logging: Page repair, system restart, media restore, and system failover," in *Synthesis Lectures on Data Management*, 2nd edition, Morgan & Claypool Publishers, 2016, pp. 1–113.

[35] G. Graefe, H. Kimura, and H. A. Kuno, "Foster b-trees," *ACM ToDS*, vol. 37, no. 3, 2012, 17:1–17:29.

[36] G. Graefe and H. A. Kuno, "Fast loads and fast queries," *Transactions on Large-Scale Data- and Knowledge-Centered Systems*, vol. 2, 2010, pp. 31–72.

[37] G. Graefe and H. A. Kuno, "Self-selecting, self-tuning, incrementally optimized indexes," *EDBT Conference*, 2010, pp. 371–381.

[38] G. Graefe and H. A. Kuno, "Definition, detection, and recovery of single-page failures, a fourth class of database failures," *PVLDB*, vol. 5, no. 7, 2012, pp. 646–655.

[39] G. Graefe, H. A. Kuno, and B. Seeger, "Self-diagnosing and self-healing indexes," *DBTest*, 2012, p. 8.

[40] G. Graefe, M. Lillibridge, H. A. Kuno, J. Tucek, and A. C. Veitch, "Controlled lock violation," *ACM SIGMOD Conference*, 2013, 85–96. Also in *Synthesis Lectures on Data Management*, Morgan & Claypool Publishers.

[41] G. Graefe, I. Petrov, T. Ivanov, and V. Marinov, "A hybrid page layout integrating PAX and NSM," *IDEAS*, 2013, pp. 86–95.

[42] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch, "In-memory performance for big data," *PVLDB*, vol. 8, no. 1, 2014, pp. 37–48.

[43] J. Gray and A. Reuter, *Transaction Processing Concepts and Techniques*. Morgan Kaufmann, 1993.

[44] T. Härder, "Observations on optimistic concurrency control schemes," *Information Systems*, vol. 9, no. 2, 1984, pp. 111–120.

[45] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," *ACM SIGMOD Conference*, 2008, pp. 981–992.

[46] B. R. Iyer, "Hardware-assisted sorting in IBM's DB2 DBMS," in *COMAD Conference*, Hyderabad, 2005.

[47] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti, "Incremental organization for data recording and warehousing," *VLDB Conference*, 1997, pp. 16–25.

[48] J. R. Jordan, J. Banerjee, and R. B. Batman, "Precision locks," *ACM SIGMOD Conference*, 1981, pp. 143–147.

[49] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "RadixSpline: A single-pass learned index," *aiDM@SIGMOD*, 2020, 5:1–5:5.

[50] T. Kraska, A. Beutel, E. H. Chi, and J. Dean, "The case for learned index structures," in *ACM SIGMOD Conference*, 2018, pp. 489–504.

[51] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM ToDS*, vol. 6, no. 2, 1981, pp. 221–226.

[52] P. L. Lehman and S. B. Yao, "Efficient locking for concurrent operations on b-trees," *ACM ToDS*, vol. 6, no. 4, 1981, pp. 650–670.

[53] H. Leslie, R. Jain, D. Birdsall, and H. Yaghmai, "Efficient search of multi-dimensional b-trees," *VLDB Conference*, 1995, pp. 710–719.

[54] D. B. Lomet, "Key range locking strategies for improved concurrency," *VLDB Conference*, 1993, pp. 655–664.

[55] D. B. Lomet, "The evolution of effective b-tree: Page organization and techniques: A personal account," *ACM SIGMOD Record*, vol. 30, no. 3, 2001, pp. 64–69.

[56] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *The VLDB Journal*, vol. 29, no. 1, 2020, pp. 393–418.

[57] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," *EuroSys*, 2012, pp. 183–196.

[58] C. Mohan, "ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on b-tree indexes," *VLDB Conference*, 1990, pp. 392–405.

[59] C. Mohan and F. E. Levine, "ARIES/IM: An efficient and high concurrency index management method using write-ahead logging," *ACM SIGMOD Conf.*, 1992, pp. 371–380.

[60] T. Neumann, T. Mühlbauer, and A. Kemper, "Fast serializable multi-version concurrency control for main-memory database systems," *ACM SIGMOD Conference*, 2015, pp. 677–689.

[61] C. Nyberg, T. Barclay, Z. Cvetanovic, J. Gray, and D. B. Lomet, "AlphaSort: A cache-sensitive parallel external sort," *The VLDB Journal*, vol. 4, no. 4, 1995, pp. 603–627.

[62] P. E. O'Neil, "The SB-tree: An index-sequential structure for high-performance sequential access," *Acta Informatica*, vol. 29, no. 3, 1992, pp. 241–265.

[63] P. E. O'Neil, E. Cheng, D. Gawlick, and E. J. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, 1996, pp. 351–385.

[64] D. A. Patterson, G. A. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (RAID)," *ACM SIGMOD Conference*, 1988, pp. 109–116.

[65] P. V. Sandt, Y. Chronis, and J. M. Patel, "Efficiently searching in-memory sorted arrays: Revenge of the interpolation search?" *ACM SIGMOD Conference*, 2019, pp. 36–53.

[66] C. Sauer, G. Graefe, and T. Härder, "Single-pass restore after a media failure," *BTW*, 2015, pp. 217–236.

[67] R. Sears and R. Ramakrishnan, "bLSM: A general purpose log-structured merge tree," *ACM SIGMOD Conference*, 2012, pp. 217–228.

[68] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," *ACM SIGMOD Conf.*, 1979, pp. 23–34.

[69] D. G. Severance and G. M. Lohman, "Differential files: Their application to the maintenance of large databases," *ACM Transactions on Database Systems*, vol. 1, no. 3, 1976, pp. 256–267.

[70] A. Verbitski, A. Gupta, D. Saha, M. Brahmadesam, K. Gupta, R. Mittal, S. Krishnamurthy, S. Maurice, T. Kharatishvili, and X. Bao, "Amazon Aurora: Design considerations for high throughput cloud-native relational databases," *ACM SIGMOD Conference*, 2017, pp. 1041–1052.

[71] G. Weikum and G. Vossen, *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

[72] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber, "Designing a succinct secondary indexing mechanism by exploiting column correlations," *ACM SIGMOD Conference*, 2019, pp. 1223–1240.

[73] H. Zhang, X. Liu, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo, "Order-preserving key compression for in-memory search trees," *ACM SIGMOD Conference*, 2020, pp. 1601–1615.