A survey of B-tree locking techniques

Goetz Graefe

HP Laboratories
HPL-2010-9

Keyword(s):
storage systems, databases, B-tree, indexes, concurrency control, locking

Abstract:
B-trees have been ubiquitous in database management systems for several decades, and they are used in other storage systems as well. Their basic structure and basic operations are well and widely understood including search, insertion, and deletion. Concurrency control of operations in B-trees, however, is perceived as a difficult subject with many subtleties and special cases. The purpose of this survey is to clarify, simplify, and structure the topic of concurrency control in B-trees by dividing it into two sub-topics and exploring each of them in depth.

External Posting Date: January 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: January 21, 2010 [Fulltext]

To be published in ACM Transactions on Database Systems (TODS), Volume 35, Issue 2, 2010.

A survey of B-tree locking techniques

Goetz Graefe
Hewlett-Packard Laboratories

Abstract

B-trees have been ubiquitous in database management systems for several decades, and they are used in other storage systems as well. Their basic structure and basic operations are well and widely understood including search, insertion, and deletion. Concurrency control of operations in B-trees, however, is perceived as a difficult subject with many subtleties and special cases. The purpose of this survey is to clarify, simplify, and structure the topic of concurrency control in B-trees by dividing it into two sub-topics and exploring each of them in depth.

1 Introduction

B-tree indexes [Bayer and McCreight 1972] have been called ubiquitous more than a quarter of a century ago [Comer 1979], and they have since become ever more ubiquitous. Gray and Reuter asserted that "B-trees are by far the most important access path structure in database and file systems" [Gray and Reuter 1993]. In spite of many innovative proposals and prototypes for alternatives to B-tree indexes, this statement remains true today.

As with all indexes, their basic function is to map search keys to associated information. In addition to exact-match lookup, B-trees efficiently support range queries and they enable sort-based query execution algorithms such as merge join without explicit sort operation. More recently, B-tree indexes have been extended to support multi-dimensional data and queries by using space-filling curves, for example, the Z-order in UB-trees [Bayer 1997, Ramsak et al. 2000].

The basic data structure and algorithms for B-trees are well understood. Search, insertion, and deletion are often implemented by college students as programming exercises, including split and merge operations for leaves and interior nodes. Accessing B-tree nodes on disk storage using a buffer pool adds fairly little to the programming effort. Variable-length records within fixed-length B-tree nodes add moderate complexity to the code, mostly book-keeping for lengths, offsets, and free space. It is far more challenging to enable correct multi-threaded execution and even transactional execution of B-tree operations.

Database servers usually run in many threads to serve many users as well as to exploit multiple processor cores and, using asynchronous I/O, many disks. Even for single-threaded applications, e.g., on personal computing devices, asynchronous activities for database maintenance and index tuning require concurrent threads and thus concurrency control in B-tree indexes.

The plethora of terms associated with concurrency control in B-trees may seem daunting, including row-level locking, key value locking, key range locking, lock coupling, latching, latch coupling, and crabbing, the last term applied to both root-to-leaf searches and leaf-to-leaf scans. A starting point for clarifying and simplifying the topic is to distinguish protection of the B-tree structure from protection of the B-tree contents, and to distinguish separation of threads against one another from separation of user transactions against one another. These distinctions are central to the treatment of the topic here. Confusion between these forms of concurrency control in B-trees unnecessarily increases the efforts for developer education, code maintenance, test development, test execution, and defect isolation and repair.

The foundations of B-tree locking are the well-known transaction concept, including multi-level transactions [Weikum 1991], open nested transactions [Ni et al. 2007, Weikum and Schek 1992], and pessimistic concurrency control, i.e., locking, rather than optimistic concurrency control [Kung and Robinson 1981]. Multiple locking concepts and techniques are discussed here, including phantom protection, predicate locks, precision locks, key value locking, key range locking, multi-granularity locking, hierarchical locking, and intention locks.

No actual system works precisely as described here. There are many reasons for this fact; one is that code for concurrency control is complex and rarely touched, such that many concepts and implementations are years or even decades old.

1.1 Historical background

Although the original B-tree design employed data items and their keys as separators in the nodes above the B-tree leaves, even the earliest work on concurrency control in B-trees relied on all data items being in the leaves, with separator keys in the upper B-tree levels serving only to guide searches without carrying actual information contents.

Bayer and Schkolnick [1977] presented multiple locking (latching) protocols for B*-trees (all data records in the leaves, merely separator or "reference" keys in upper nodes) that combined high concurrency with deadlock avoidance. Their approach for insertion and deletion is based on deciding during a root-to-leaf traversal whether a node is "safe" from splitting (during an insertion) or merging (during a deletion), and on retaining appropriate locks (latches) for ancestors of unsafe nodes. Bernstein et al. [1987] cover this and other early protocols for tree locking. Unfortunately, this definition of safety does not work well with variable-size records and keys.

IBM's System R project explored many transaction management techniques, including transaction isolation levels and lock duration, predicate locking and key locking, multi-granularity and hierarchical locking, etc. These techniques have been adapted and refined in many research and product efforts since then. Research into multi-level transactions [Weikum 1991] and into open nested transactions [Moss 2006] enables crisp separation of locks and latches, the former protecting database contents against conflicts among transactions and the latter protecting in-memory data structures against conflicts among concurrent threads.

Mohan's ARIES/KVL design [Mohan 1990] explicitly separates locks and latches in a similar way as described here, i.e., logical database contents versus "structure maintenance" in a B-tree. A key value lock covers both a gap between two B-tree keys and the upper boundary key. In non-unique indexes, an intention lock on a key value permits operations on all rows with the same value in the indexed column. In contrast, other designs include the row identifier in the unique lock identifier and thus do not need to distinguish between unique and non-unique indexes. Gray and Reuter's book [1993] describes row-level locking in B-trees based primarily on ARIES/KVL.

In order to reduce the number of locks required during index-to-index navigation, e.g., searching for a row in a non-clustered index followed by fetching additional columns in a clustered index, a single lock in ARIES/IM [Mohan and Levine 1992] covers all index entries for a logical row even in the presence of multiple indexes. Srinivasan and Carey [1991] compared the performance of various locking techniques.

Lomet's design for key range locking [Lomet 1993] attempts to adapt hierarchical and multi-granularity locking to keys and half-open intervals but requires additional lock modes, e.g., a 'range insert' mode, to achieve the desired concurrency. Graefe's design [2007] applies traditional hierarchical locking more rigorously to keys and gaps (open intervals) between keys, employs ghost (pseudo-deleted) records during insertion as well as during deletion, and achieves more concurrency with fewer special cases. The same paper also outlines hierarchical locking in B-tree structures (key range locking on separator keys) and in B-tree keys (key range locking on fixed-length prefixes of compound keys).

Numerous text books on database systems cover concurrency control in index structures, with various degrees of generality, completeness, and clarity. Weikum and Vossen's book [2002] comes closest to our perspective on the material, focusing on a page layer and an access layer. The discussion here resonates with their treatment but focuses more specifically on common implementation techniques using latching for data structures and key range locking for contents protection.

Some researchers have used "increment" locks as examples for general locking concepts, e.g., Korth [1983]. Others have argued that increment locks, differently than write locks, should enable higher concurrency as increment operations commute. In fact, techniques very similar to increment locks have been used in real high-performance systems [Gawlick and Kinkade 1985] and described in a more general and conceptual way with additional "escrow" guarantees [O'Neil 1986]. A more recent effort has focused on integrating increment locks with key range locking in B-tree indexes including maximal concurrency during creation and removal of summary records [Graefe and Zwilling 2004].

In spite of these multiple designs and the many thoughtful variants found in actual implementations, we believe with Gray and Reuter [1993] that "the last word on how to control concurrency on B-trees optimally has not been spoken yet."

1.2 Overview

The following two short sections clarify assumptions and define the two forms of B-tree locking that are often confused. The next two sections cover these two forms of locking in depth, followed by detailed discussions of a variety of locking techniques proposed for B-tree indexes. Future directions, summary, and conclusions close this paper.

2 Preliminaries

Most work on concurrency control and recovery in B-trees assumes what Bayer and Schkolnick [1977] call B*-trees and what Comer [1979] calls B+-trees, i.e., all data records are in leaf nodes and keys in non-leaf or "interior" nodes act merely as separators enabling search and other operations but not carrying logical database contents. We follow this tradition here and ignore the original design of B-trees with data records in both leaves and interior nodes.

We also ignore many other variations of B-trees here. This includes what Comer, following Knuth, calls B*-trees, i.e., attempting to merge an overflowing node with a sibling rather than splitting it immediately. We ignore whether or not underflow is recognized and acted upon by load balancing and merging nodes, whether or not empty nodes are removed immediately or ever, whether or not leaf nodes form a singly or doubly linked list using physical pointers (page identifiers) or logical boundaries (fence keys equal to separators posted in the parent node during a split), whether suffix truncation is employed when posting a separator key [Bayer and Unterauer 1977], whether prefix truncation or any other compression is employed on each page, and the type of information associated with B-tree keys. Most of these issues have little or no bearing on locking in B-trees, with the exception of sibling pointers, as indicated below where appropriate.

Most concurrency control schemes distinguish between reading and writing, the latter covering any form of update or state change. These actions are protected by shared and exclusive locks, abbreviated S and X locks hereafter (some sections also use N as a lock mode indicating no lock). For these, the standard lock table is shown in Figure 1.

	S	*X*
N	ok	ok
S	ok	
X		

Figure 1. Minimal lock table.

One axis indicates the lock already held by one thread or transaction and the other axis indicates the lock requested by another thread or transaction. Some systems employ non-symmetric lock tables; if in doubt, the left column indicates the lock already held and the top row indicates the lock requested. Obviously, if no lock is held yet on a resource, any lock request may succeed.

3 Two forms of B-tree locking

B-tree locking, or locking in B-tree indexes, means two things. First, it means concurrency control among concurrent database transactions querying or modifying database contents and its representation in B-tree indexes. Second, it means concurrency control among concurrent threads modifying the B-tree data structure in memory, including in particular images of disk-based B-tree nodes in the buffer pool.

These two aspects have not always been separated cleanly. Their difference becomes very apparent when a single database request is processed by multiple parallel threads. Specifically, two threads within the same transaction must "see" the same database contents, the same count of rows in a table, etc. This includes one thread "seeing" updates applied by the other thread. While one thread splits a B-tree node, however, the other thread must not observe intermediate and incomplete data structures. The difference also becomes apparent in the opposite case when a single execution thread serves multiple transactions.

3.1 Locks and latches

These two purposes are usually accomplished by two different mechanisms, locks and latches. Unfortunately, the literature on operating systems and programming environments usually uses the term locks for the mechanisms that in database systems are called latches. Figure 2 summarizes their differences.

Locks separate transactions using read and write locks on pages, on B-tree keys, or even on gaps (open intervals) between keys. The latter two methods are called key value locking and key range locking. Key range locking is a form of predicate locking that uses actual key values in the B-tree and the B-tree's sort order to define predicates. By default, locks participate in deadlock detection and are held until end-of-transaction. Locks also support sophisticated scheduling, e.g., using queues for pending lock requests and delaying new lock acquisitions in favor of lock conversions, e.g., an existing shared lock to an exclusive lock. This level of sophistication makes lock acquisition and release fairly expensive, often thousands of CPU cycles, some of those due to cache faults in the lock manager's hash table and linked lists.

	Locks	*Latches*
Separate …	User transactions	Threads
Protect …	Database contents	In-memory data structures
During …	Entire transactions	Critical sections
Modes …	Shared, exclusive, update, intention, escrow, schema, etc.	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, "lock leveling"
Kept in …	Lock manager's hash table	Protected data structure

Figure 2. Locks and latches.

Latches separate threads accessing B-tree pages, the buffer pool's management tables, and all other in-memory data structures shared among multiple threads. Since the lock manager's hash table is one of the data structures shared by many threads, latches are required while inspecting or modifying a database system's lock information. With respect to shared data structures, even threads of the same user transaction conflict if one thread requires a write latch. Latches are held only during a critical section, i.e., while a data structure is read or updated. Deadlocks are avoided by appropriate coding disciplines, e.g., requesting multiple latches in carefully designed sequences. Deadlock resolution requires a facility to roll back prior actions, whereas deadlock avoidance does not. Thus, deadlock avoidance is more appropriate for latches, which are designed for minimal overhead and maximal performance and scalability. Latch acquisition and release may require tens of instructions only, usually with no additional cache faults since a latch can be embedded in the data structure it protects. For images of disk pages in the buffer pool, the latch can be embedded in the descriptor structure that also contains the page identifier etc.

The difference between locking and latching also becomes apparent in concurrency control for non-clustered indexes, i.e., redundant indexes that point into non-redundant storage structures. Data-only locking makes no attempt to separate transactions using the non-clustered index and its keys. All inter-transaction concurrency control is achieved by locks on non-redundant data items, e.g., record identifiers in the table's heap structure. Latches, on the other hand, are required for any in-memory data structure touched by multiple concurrent threads, including of course the nodes of the non-clustered index.

3.2 Recovery and B-tree locking

The difference between latches and locks is important not only during normal transaction processing but also during recovery from a system crash. Considering the requirements for latching and locking during crash recovery may further illuminate their difference.

Latches and locks also differ both during system recovery and while waiting for the decision of a global transaction coordinator. While waiting, no latches are required, but retaining locks is essential to guarantee a local transaction's ability to abide by the global coordinator's final decision. During recovery without concurrent execution of new transactions, locks are not required, because concurrency control during forward processing prior to the system crash already ensured that active transactions do not conflict. Latches, however, are as important during recovery as during normal forward processing if recovery employs multiple threads and shared data structures such as the buffer pool.

The separation of locks and latches does not determine whether an operation must be logged for recovery purposes. Logging is required if disk contents change, whether or not the logical database contents changes. Note that physiological logging refers to records by their identifier within their page and not by their byte location; thus, movement of records within a page is one of the few operations that do not require auditing in the recovery log. Compaction of free space and record space within a page does not require log-

ging, unless invalid records are removed in such a way that remaining records change their identifiers, e.g., their slot numbers within a page.

Locks must be retained during transaction rollback, although they could be released incrementally, e.g., during a partial rollback to a transaction savepoint. During recovery from a system failure, lock acquisition during log analysis permits resuming transaction processing while repeating logged actions and applying compensating actions of aborted transactions. These locks need not be the same ones acquired during the original transactions. For example, if hierarchical locking (see below) is supported in the system, locks during recovery may be smaller or larger, as long as they cover the actions performed by the transaction. If lock escalation and de-escalation are supported, these techniques can be employed even by transactions currently in recovery. Similarly, lock escalation and de-escalation may be invoked while a local transaction waits for a global decision by a transaction coordinator.

The operations protected by latches only (and not by locks) are those that modify a B-tree's structure without modifying its logical contents. Typical examples are splitting a B-tree node and balancing the load among neighboring B-tree nodes. Other examples that affect entire B-tree indexes include compression, defragmentation, and other forms of reorganization. Examples that affect less than an entire node are creation and removal of ghost records (see below). As the logical database contents is the same whether such an operation is completed or not, recovery from a system failure during such an operation may either roll the operation back or complete it. Completion or "forward recovery" of such operations may be faster and require less log volume than rollback. In order to prepare an operation for forward recovery, pre-allocation of all required resources is essential, most notably of sufficient disk space. In general, forward recovery seems more promising for deletion or deallocation operations, e.g., dropping a B-tree index or erasing a ghost record, than for creation or allocation operations.

3.3 Lock-free B-trees

Occasionally, one sees claims or descriptions of database indexes implemented "lock-free." This might refer to either of the two forms of B-tree locking. For example, it might mean that the in-memory format is such that pessimistic concurrency control in form of latches is not required. Multiple in-memory copies and atomic updates of in-memory pointers might enable such an implementation. Nonetheless, user transactions and their access to data must still be coordinated, typically with locks such as key range locking on B-tree entries.

On the other hand, a lock-free B-tree implementation might refer to avoidance of locks for coordination of user transactions. A typical implementation mechanism would require multiple versions of records based on multi-version concurrency control. However, creation, update, and removal of versions must still be coordinated for the data structure and its in-memory image. In other words, even if conflicts among read transactions and write transactions can be removed by means of versions rather than traditional database locks, modifications of the data structure still requires latches.

In the extreme case, a lock-free implementation of B-tree indexes might avoid both forms of locking in B-trees. Without further explanation, however, one has no way of knowing what specifically is meant by a lock-free B-tree implementation.

3.4 Summary

In summary, latching and locking serve different functions for B-tree indexes in database systems. Accordingly, they use different implementation primitives. The difference is starkly visible not only during normal transaction processing but also during "offline" activities such as crash recovery.

4 Protecting a B-tree's physical structure

If multiple threads may touch an in-memory data structure, their concurrent accesses must be coordinated. This is equally true whether these data structures always reside in memory, e.g., the look-up table in a buffer pool or pages in an in-memory database, or reside in memory only temporarily, e.g., images of disk pages in a buffer pool. Multiple threads are required if multiple concurrent activities are desired, e.g., in shared-memory machine or in a modern many-core processor.

The simplest form of a latch is a "mutex" (mutual exclusion lock). Any access precludes concurrent access. For data structures that change constantly, this might be an appropriate mechanism; for disk pages

that remain unchanged during query processing, exclusion of all concurrent activities should be reserved for updates. Latches with shared and exclusive (read and write) modes are common in database systems.

Since latches protect in-memory data structures, the data structure representing the latch can be embedded in the data structure being protected. Latch duration is usually very short and amenable to hardware support and encapsulation in transactional memory [Larus and Rajwar 2006]. This is in contrast to locks, discussed below, that protect the logical database contents. Since locks often protect data not even present in memory, and sometimes not even in the database, there is no in-memory data structure within which the data structure representing the lock can be embedded. Thus, database systems employ lock tables. Since a lock table is an in-memory data structure, concurrent access to a lock table and its components is protected by latches embedded in the protected data structures.

4.1 Issues

Latches ensure the consistency of data structures when accessed by multiple concurrent threads. They solve several problems that are similar to each other but nonetheless lend themselves to different solutions. All these issues are about the consistency of in-memory data structures, including images of disk pages in the buffer pool.

First, a page image in the buffer pool must not be modified (written) by one thread while it is interpreted (read) by another thread. This issue does not differ from other critical sections for shared data structures in multi-threaded code, including critical sections in a database system protecting the lock manager's hash table or the buffer pool's table of contents.

Second, while following a pointer (page identifier) from one page to another, e.g., from a parent node to a child node in a B-tree index, the pointer must not be invalidated by another thread, e.g., by deallocating a child page or balancing the load among neighboring pages. This issue requires more refined solutions than the first issue above because it is not advisable to retain a latch (or otherwise extend a critical section) while performing I/O (e.g., to fetch a child page into the buffer pool).

Third, "pointer chasing" applies not only to parent-child pointers but also to neighbor pointers, e.g., in a chain of leaf pages during a range scan or while searching for the key to lock in key range locking (see below). Concurrent query execution plans, transactions, and threads may perform ascending and descending index scans, which could lead to deadlocks. Recall that latches usually rely on developer discipline for deadlock avoidance, not on automatic deadlock detection and resolution.

Fourth, during a B-tree insertion, a child node may overflow and require an insertion into its parent node, which may thereupon also overflow and require an insertion into the child's grandparent node. In the most extreme case, the B-tree's old root node overflow, must split, and be replaced by a new root node. Going back from the leaf towards the B-tree root works well in single-threaded B-tree implementations, but it introduces the danger of deadlocks, yet latches rely on deadlock avoidance rather than deadlock detection and resolution.

For the first issue above, database systems employ latches that differ from the simplest implementations of critical sections and mutual exclusion only by the distinction between read-only latches and read-write latches, i.e., shared or exclusive access. As with read and write locks protecting database contents, starvation needs to be avoided if new readers are admitted while a writer already waits for active readers to release the latch. Update latches (similar to update locks) may be used, but the other modes well-known for lock management are usually not adopted in database latches.

4.2 Lock coupling

The second issue above requires retaining the latch on the parent node until the child node is latched. This technique is traditionally called "lock coupling" but a better term is "latch coupling" in the context of transactional database systems. The lookup operation in the buffer pool requires latches to protect the buffer manager's management tables, and proper "lock leveling" (latch leveling) is required to guarantee that there can be no deadlock due to latches on data pages and the buffer pool.

If the child node is not readily available in the buffer pool and thus requires relatively slow I/O, the latch on the parent node should be released while reading the child node from disk. Lock coupling can be realized by holding a latch on the descriptor of the needed page in the buffer pool. Otherwise, the I/O should be followed by a new root-to-leaf traversal to protect against B-tree changes in the meantime. Restarting the root-to-leaf traversal may seem expensive but it is often possible to resume the prior search by

verifying, e.g., based on the log sequence number on each page, that the parent page has not been modified while the child page was fetched into the buffer pool.

The third issue above is similar to the second, with two differences. On the positive side, asynchronous read-ahead may alleviate the frequency of buffer faults. Deep read-ahead in B-tree indexes usually cannot rely on the neighbor pointers but requires prefetch based on child pointers in leaves' parent nodes or even the grandparent nodes. On the negative side, deadlock avoidance among scans in opposite directions requires that latch acquisition code provides an immediate failure mode. A latch acquisition request can return a failure immediately rather than waiting for the latch to become available. If such a failure occurs during forward or backward index traversal, the scan must release the leaf page currently latched to let the conflicting scan proceed, and reposition itself using a new root-to-leaf search.

The fourth issue above is the most complex one. It affects all updates, including insertion, deletion, and even record updates, the latter if length changes in variable-length records can lead to nodes splitting or merging. The most naïve approach, latching an entire B-tree with a single exclusive latch, is obviously not practical in multi-threaded servers, and all approaches below latch individual B-tree nodes.

One approach latches all nodes in exclusive mode during the root-to-leaf traversal while searching for an affected leaf. The obvious problem with this approach is that it can create a concurrency bottleneck, particularly at a B-tree's root page. Another approach performs the root-to-leaf search using shared latches and attempts an upgrade to an exclusive latch when necessary. When the upgrade can be granted without the danger of deadlock, this approach works well. Since it might fail, however, it cannot be the only method in a B-tree implementation. Thus, it might not be implemented at all in order to minimize code volume and complexity. A third approach reserves nodes using "update" or "upgrade" latches in addition to traditional shared and exclusive latches. Update locks or latches are compatible with shared locks or latches and thus do not impede readers or B-tree searches, but they are not compatible with each other and thus a B-tree's root node can still be a bottleneck for multiple updates.

A refinement of the three earlier approaches retains latches on nodes along its root-to-leaf search only until a lower, less-than-full node guarantees that split operations will not propagate up the tree beyond the lower node. Since most nodes are less than full, most insertion operations will latch no nodes in addition to the current one. On the other hand, variable-length B-tree records and variable-length separator keys seem to make it difficult or impossible to decide reliably how much free space is required for the desired guarantee. Interestingly, this problem can be solved by deciding, before releasing the latch on the parent, which separator key would be posted in the parent if the child were to split as part of the current insertion. If it is desirable to release the parent latch before such inspection of the child node, a heuristic method may consider the lengths of existing separators in the parent, specifically their average and standard deviation.

A fourth approach splits nodes proactively during a root-to-leaf traversal for an insertion. This method avoids both the bottleneck of the first approach and the failure point (upgrading from a shared to an exclusive latch) of the second approach. Its disadvantage is that it wastes some space by splitting earlier than truly required, and more importantly that it may be impossible to split proactively in all cases in B-trees with variable-length records and keys.

A fifth approach protects its initial root-to-leaf search with shared latches, aborts this search when a node requires splitting, restarts a new one, and upon reaching the node requiring a split, acquires an exclusive latch and performs the split. This approach can benefit from resuming a root-to-leaf traversal rather than restarting it at the root, as discussed earlier.

4.3 B^{link}-trees

An entirely different approach relaxes the data structure constraints of B-trees and divides a node split into two independent steps, as follows. Each node may have a high fence key and a pointer to its right neighbor, thus the name B^{link}-trees [Lehman and Yao 1981]. If a node's high fence and pointer are actually used, the right neighbor is not yet referenced in the node's parent. In other words, a single key interval and its associated child pointer in the parent node really refer to two nodes in this case. A root-to-leaf search, upon reaching a node, must first compare the sought key with a node's high fence and proceed to the right neighbor if the sought key is higher than the fence key. The first step of splitting a node creates the high fence key and a new right neighbor. The second, independent step posts the high fence key in the parent. The second step can be made a side effect of any future root-to-leaf traversal, should happen as soon as possible, yet may be delayed beyond a system reboot or even a crash and its recovery without data loss or inconsistency of the on-disk data structures. The advantage of B^{link}-trees is that allocation of a new node

and its initial introduction into the B-tree is a local step, affecting only one preexisting node. The disadvantages are that search may be a bit less efficient, a solution is needed to prevent long linked lists among neighbor nodes during periods of high insertion rates, and verification of a B-tree's structural consistency is more complex and perhaps less efficient.

Figure 3 illustrates a state that is not possible in a standard B-tree but is a correct intermediate state in a B^{link}-tree. "Correct" here means that search and update algorithms must cope with this state and that a database utility that verifies correct on-disk data structures must not report an error. In the original state, the parent node has three children. Note that these children might be leaves of interior nodes, and the parent might be the B-tree root or an interior node. The first step is to split a child, resulting in the intermediate state shown in Figure 3. The second step later places a fourth child pointer into the parent and abandons the neighbor pointer, unless neighbor pointers are required in a specific implementation of B-trees. In fact, a third step might erase the neighbor pointer value from the child node.

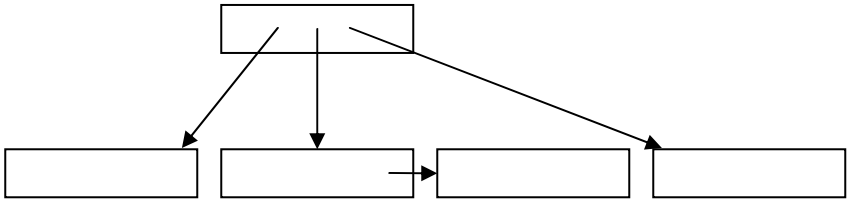

Figure 3. Intermediate state in a B^{link}-tree.

This process can be reversed in order to enable removal of B-tree nodes [Lomet 2004]. The first step creates a neighbor pointer, the second erases the child pointer in the parent node, and the third step merges the removal victim with its neighbor node.

Long linked list due to multiple splits can be prevented by restricting the split operation to nodes pointed to by the appropriate parent node. These and further details of B^{link}-trees have recently been described in a detailed paper [Jaluta et al. 2005].

4.4 Load balancing and reorganization

In many implementations, removal of nodes, load balancing among nodes, etc. are omitted in user transactions and left to asynchronous facilities. These may be invoked immediately after a user transaction, guided to a specific B-tree node by information left behind by the user transaction, or they may be invoked occasionally to scan and optimize an entire B-tree, index, table, or database. Actually, even node splits could be organized in this way, ensuring that user transactions can always add one more entry to any B-tree node.

All these operations are contents-neutral; the logical database contents does not change, only its representation. Thus, once the affected database pages are present in the buffer pool, these changes require concurrency control merely for the data structures. In other words, these operations require latches but no locks. Of course, since subsequent user transactions may log changes that refer to changes due to such B-tree optimizations, these optimizations must be reflected in the recovery log.

As an aside, there are corresponding optimizations in logging and recovery. One can avoid logging the page contents by careful write ordering [Gray and Reuter 1993], and one can employ "forward recovery" to complete a reorganization action after system restart instead of rollback [Zou and Salzberg 1996].

4.5 Summary

In summary, latches coordinate multiple execution threads while they access and modify database contents and its representation. While cached in the database buffer pool, these data structures require the same coordination and protection as data structures representing server state, e.g., the table of contents of the buffer pool, the list of active transactions, or the hash table in the lock manager.

The implementation of latches is optimized for their frequent use. In particular, modes beyond shared and exclusive are avoided, and deadlock avoidance is favored over deadlock detection. In complex data structures such as B-trees, careful designs such as crabbing and B^{link}-trees combine high concurrency and deadlock avoidance.

Latches should be held only for very short periods. None should be held during disk I/O. Thus, all required data structures should be loaded and pinned in the buffer pool before a latched operation begins.

In the following, we focus on coordination among transactions and assume that the required data structures have the necessary coordination and protection by latches.

5 Protecting a B-tree's logical contents

Locks separate transactions reading and modifying database contents. For serializability, read locks are retained until end-of-transaction. Write locks are always retained until end-of-transaction in order to ensure the ability to roll back all changes if the transaction aborts. Weaker retention schemes exist for both read locks and write locks. Shorter read locks lead to weak transaction isolation levels; shorter write locks may lead to cascading aborts among transactions that read and modify the same database item.

In addition, serializability requires locking not only the presence but also the absence of data. For example, if a range scan finds 10 rows, a second execution of the same range scan must again find 10 rows, and an insertion by another transaction must be prevented. In B-tree indexes, this is usually enforced by key range locking, i.e., locks that cover a key value and the gap to the next key value.

Modern designs for key range locking are based on hierarchical or multi-granularity locking. Multi-granularity locking is often explained and illustrated in terms of pages and files but it can be applied in many other ways. If the granules form a strict hierarchy, not a directed acyclic graph, the term hierarchical locking can be used. In modern B-tree implementations and their locks on keys, hierarchical locking is used to protect individual B-tree entries, the gaps or open intervals between existing B-tree keys, and half-open intervals comprising a B-tree entry and one adjoining open interval.

A fine granularity of locking is required most often at hot spots, e.g., "popular" insertion points such as the high end of a time-organized B-tree. Coarse locks such as intention locks on a database, table, or index may be retained from one transaction to another. Requests for conflicting locks may employ an immediate notification or commit processing of the holding transaction may verify that the wait queue for the lock remains empty. For the latter design, a successful lock request should return a pointer to the lock data structure in order to avoid a subsequent unnecessary search in the lock manager's hash table.

Another useful implementation technique separates the "test" and "set" functions within lock acquisition, i.e., verification that a certain lock could be acquired and insertion of a lock into the lock manager's hash table. Test without set is equivalent to what has been called "instant locks," e.g., by Gray and Reuter [1993]; set without test is useful during lock de-escalation, lock re-acquisition during system recovery, and other situations in which lock conflicts can be ruled out without testing.

5.1 Key range locking

The terms key value locking and key range locking are often used interchangeably. Key range locking is a special form of predicate locking [Eswaran et al. 1976]. Neither pure predicate locking nor the more practical precision locking [Jordan et al. 1981] has been adopted in major products. In key range locking, the predicates are defined by intervals in the sort order of the B-tree. Interval boundaries are the key values currently existing in the B-tree. The usual form are half-open intervals including the gap between two neighboring keys and one of the end points, with "next-key locking" perhaps more common than "prior-key locking." The names describe the key to be locked in order to protect the gap between two neighboring keys. Next-key locking requires the ability to lock an artificial key value "+∞". Prior-key locking gets by with locking the NULL value, assuming this is the lowest possible value in the B-tree's sort order.

In the simplest form of key range locking, a key and the gap to the neighbor are locked as a unit. An exclusive lock is required for any form of update of this unit, including modifying non-key fields of the record, deletion of the key, insertion of a new key into the gap, etc. Deletion of a key requires a lock on both the old key and its neighbor; the latter is required to ensure the ability to re-insert the key in case of transaction rollback.

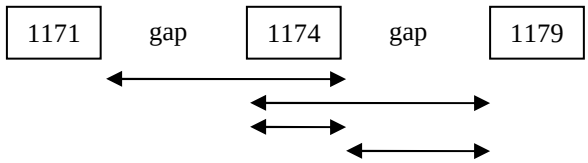

Figure 4. Possible lock scopes.

Figure 4 summarizes the possible scopes in key range locking in a B-tree leaf containing three records in the key range 1170 to 1180. A lock on key value 1174 might have any of the ranges indicated by arrows. The first arrow illustrates traditional next-key locking. The second arrow indicates prior-key locking. The third arrow shows a lock limited to the key value itself, without coverage of either one of the neighboring gaps. Thus, this lock cannot guarantee absence of a key for a transaction's duration, e.g., key value 1176, and it therefore cannot guarantee serializability. The fourth lock scope complements the key value lock; it can guarantee absence of a key without locking an existing key. While one transaction holds a lock on key value 1174 as shown in the fourth arrow, a second transaction may update the record with key value 1174, except of course the record's key value. The second transaction must not erase the record or the key, however, until the first transaction releases its lock. Figure 4 could show a fifth lock scope that covers the gap preceding the locked key; it is omitted because it might confuse the discussion below.

High rates of insertion can create a hotspot at the "right edge" of a B-tree index on an attribute correlated with time. With next-key locking, one solution verifies the ability to acquire a lock on $+\infty$ (infinity) but does not actually retain it. Such "instant locks" violate two-phase locking but work correctly if a single acquisition of the page latch protects both verification of the lock and creation of the new key on the page.

Latches must be managed carefully in key range locking if lockable resources are defined by keys that may be deleted if not protected. Until the lock request is inserted into the lock manager's data structures, the latch on the data structure in the buffer pool is required to ensure the existence of the key value. On the other hand, if a lock cannot be granted immediately, the thread should not hold a latch while the transaction waits. Thus, after waiting for a key value lock, a transaction must repeat its root-to-leaf search for the key.

5.2 Key range locking and ghost records

In many B-tree implementations, the user transaction requesting a deletion does not actually erase the record. Instead, it merely marks the record as invalid, "pseudo-deleted," or a "ghost record." For this purpose, all records include a "ghost bit" in their record headers that must be inspected in every query such that ghost records do not contribute to query results. Insertion of a new B-tree key for which a ghost record already exists is turned into an update of that pre-existing record. After a valid record has been turned into a ghost record in a user transaction's deletion, space reclamation happens in an asynchronous clean-up transaction. Until then, the key of a ghost record participates in concurrency control and key range locking just like the key of a valid record. During ghost removal, no transaction may hold a lock on the key. Record removal including space reclamation does not require a lock as it is merely a change in the physical representation of the database, not in database contents.

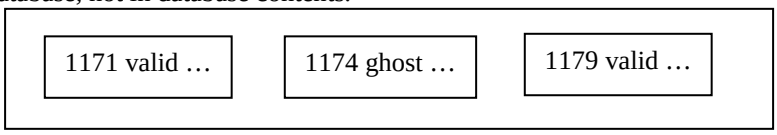

Figure 5. Intermediate state in row deletion.

Figure 5 illustrates three records in a B-tree leaf after a user transaction deleting key 1174 from the logical database contents. Each record carries a ghost bit with the possible values "valid" or "ghost." The ghost bit is not part of the key. The user transaction commits this state, leaving it to later activities to remove the record and reclaim the space within the leaf page.

The advantage of ghost records is that they avoid space allocation during transaction rollback and thus eliminate a possible rollback failure. Moreover, the user transaction need not log the deleted record, and with an appropriate implementation (fusing the log records for ghost removal and transaction commit), neither does the clean-up transaction. For B-trees with large records, e.g., clustered indexes, this can be a useful advantage. With respect to locking, since a user transaction's deletion is implemented as merely an update, it requires a lock until end-of-transaction only on the key but not on the neighboring gaps.

If a root-to-leaf search has directed an insertion operation to a specific leaf, yet the new key is higher than the highest existing key (in next-key locking; or lower than the lowest existing key in prior-key locking), then the key to be locked must be found on a neighboring leaf node. Chaining leaf pages using page identifiers speeds up this process. Nonetheless, the latch on the current page must be released while fetching the neighbor page, in order to avoid holding the latch a long time as well as creating a deadlock among multiple page latches. Alternatively, each B-tree node may include the lowest and highest possible keys. These keys are copies of the separator keys posted in the parent page during leaf splits. One of these two "fence" keys, say the highest possible key, is always a ghost record. The other fence key can be a ghost

record or it can be a valid record. Fence keys ensure that in any B-tree operation, the key to be locked can be found on the page affected by the operation. Thus, in this design, key range locking never requires navigation to neighbor nodes.

Since ghost records are not part of the logical database contents, only part of the database representation, they can be created and removed even in the presence of locks held by user transactions. For example, if a user transaction holds a key range lock on the half-open interval [10, 20), a new ghost record with value 15 can be inserted. This new key in the B-tree defines a new interval boundary for key range locking, and a range lock on value 10 now covers merely the range [10, 15). Nonetheless, the user transaction must retain its concurrency control privileges. Thus, the same operation that adds the key value 15 to the B-tree data structure must also add a lock for the user transaction on the interval [15, 20). Inversely, when a ghost record is removed, two intervals are merged and the locks held on the two intervals must also be merged. Obviously, the merged lock set must not include a lock conflict. A ghost record should not be removed while some transaction is still attempting to acquire a lock on the ghost record's key.

In addition to deletions, ghost records can also improve the performance and concurrency of insertions. Initial creation of a ghost record does not require any locks because this operation, invoked by a user transaction but not part of the user transaction, modifies merely the database representation without changing the logical database contents. Once the ghost record is in place, the user transaction merely updates an existing record including its ghost bit. The user transaction requires a lock only on the key value, not on its neighboring gaps. Thus, this two-step insertion can alleviate insertion hotspots for both prior-key locking and next-key locking. If future key values are predictable, e.g., order numbers and time values, creation of multiple ghost records at-a-time may further improve performance.

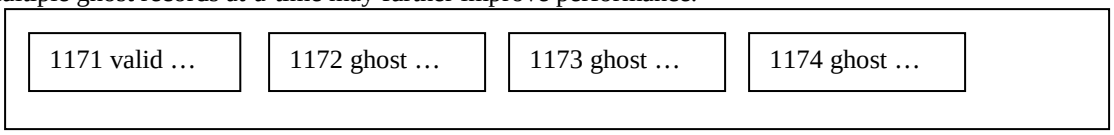

Figure 6. Insertion of multiple ghost records.

Figure 6 shows the intermediate state during a user transaction in need of appending a row with key value 1172. In preparation of future transactions, multiple ghost records are created. As a single key range lock would cover the key range from 1172 to infinity prior to this insertion, testing for conflicting locks is very fast. The user transaction triggering this insertion locks merely the first among the new ghost records, sets the ghost bit to "valid," updates the other fields in the record, and commits. If ghost records are created proactively but never used, removal and space reclamation apply just as for other ghost records.

5.3 Key range locking as hierarchical locking

When a serializable transaction locks the absence of a key, e.g., after returning the count of 0 for a query of the type "select count (*) from … where … = …", one standard design requires key range locking for the interval containing the key. Alternatively, merely the specific key can be locked, possibly after insertion as a ghost record into the appropriate index leaf page or merely into the lock manager's hash table.

Another specific example benefiting from locks on individual key values is key deletion using a ghost record. The user transaction turning a valid record into a ghost record merely needs to lock a key value, not the neighboring gaps to the neighboring keys. If forward recovery is available for ghost removal, even space reclamation requires only key value locking. Key range locking is required for key removal only to guarantee successful rollback, i.e., re-insertion of the key.

These and other cases benefit from the ability to lock a key value without locking an entire half-open interval. There are also cases in which it is desirable to lock the open interval without locking a key value. In the example above, two transactions might lock the key value [10] and the open interval (10, 20) without conflict, e.g., the query "select count (*) from … where … between 16 and 19".

Some operations such as scans, however, need to lock one or more half-open intervals in order to lock a key range much larger than a single interval between neighboring keys in a B-tree index. This could be accomplished by locking the key value and the gap between keys separately, but a single lock would be much more efficient.

Hierarchical or multi-granularity locking answers this need, even if it was not originally invented for key range locking. Figure 7 shows the traditional lock compatibility matrix for intention lock modes IS and IX and absolute locks S and X.

Any read or write operation requires a shared or exclusive lock at the appropriate granularity of locking. Before any S or IS lock is taken on any item, an IS lock is required on the larger item containing it. For example, before a page can be locked the file containing the page must be locked. X and IX locks work similarly. An IX lock also permits acquiring an S lock. An SIX lock is the combination of an S lock and an IX lock, useful when scanning a data collection searching individual items to update. The SIX lock might also be used within the lock manager to indicate that one transaction holds an S lock and another transaction holds an IX lock.

	IS	*IX*	*S*	*X*	*SIX*
IS	ok	ok	ok		ok
IX	ok	ok			
S	ok		ok		
X					
SIX	ok				

Figure 7. Lock table with intention locks.

In key range locking based on hierarchical locking, the large granularity of locking is the half-open interval; the small granularities of locking are either the key value or the open interval. This simple hierarchy permits very precise locks appropriate for the case at hand. The disadvantage of this design is that locking a key (or an open interval) requires two invocations of the lock manager, one for the intention lock on the half-open interval and one for the absolute lock on the key value.

Given that all three locks are identified by the key value, a tradeoff is possible between the number of lock modes and the number of lock manager invocations. Additional, artificial lock modes can describe combinations of locks on the half-open interval, the key value, and the open interval. Thus, a system that employs hierarchical locking for half-open interval, key value, and open interval requires no more lock management effort than one that locks only half-open intervals. Without additional effort, such a system permit additional concurrency between transactions that lock a key value and an open interval separately, e.g., to ensure absence of key values in the open interval and to update a record's non-key attributes. A record's non-key attributes include the property whether the record is a valid record or a ghost record; thus, even logical insertion and deletion are possible while another transaction locks a neighboring open interval.

Specifically, the half-open interval can be locked in S, X, IS, IX modes. The SIX mode is not required because with precisely two resources, more exact lock modes are easily possible. The key value and the open interval each can be locked in S or X modes. The new lock modes must cover all possible combinations of S, X, or N (no lock) modes of precisely two resources, the key value and the open interval. The intention locks IS and IX can remain implied. For example, if the key value is locked in X mode, the half-open interval is implicitly locked in IX mode; if the key value is locked in S mode and the open interval in X mode, the implied lock on the half-open interval containing both is the IX mode. Locks can readily be identified using two lock modes, one for the key value and one for the open interval. Assuming previous-key locking, a SN lock protects a key value in S mode and leaves the following open interval unlocked. A NS lock leaves the key unlocked but locks the open interval. This lock mode can be used for phantom protection as required for true serializability.

	NS	*NX*	*SN*	*S*	*SX*	*XN*	*XS*	*X*
NS	ok		ok	ok		ok	ok	
NX			ok			ok		
SN	ok	ok	ok	ok	ok			
S	ok		ok	ok				
SX			ok					
XN	ok	ok						
XS	ok							
X								

Figure 8. Lock table with combined lock modes.

Figure 8 shows the lock compatibility matrix. It can be derived very simply by checking for compatibility of both the first and the second components. For example, XS is compatible with NS because X is compatible with N and S is compatible with S. Single-letter locks are equivalent to using the same letter twice, but there is no benefit in introducing more lock modes than absolutely necessary.

5.4 Locking in non-unique indexes

If entries in a non-clustered index are not unique, multiple row identifiers may be associated with each value of the search key. Even thousands of record identifiers per key value are possible due to a single frequent key value or due to attributes with few distinct values. In non-unique indexes, key value locking may lock each value (and its entire cluster of row identifiers) or it may lock each unique pair of value and row identifier. The former saves lock requests in search queries, while the latter may permit higher concurrency during updates. For high concurrency in the former design, intention locks may be applied to values. Depending on the details of the design, it may not be required to lock individual row identifiers if those are already locked in the table to which the non-clustered index belongs.

A hierarchical design might permit locks for both unique values of the user-defined index key and for individual entries made unique by including the record identifier. Key range locking seems appropriate for both levels in the hierarchy, and therefore it seems sensible to use the lock modes of Figure 8 for both levels. This is a special case of the hierarchical locking in B-tree keys [Graefe 2007].

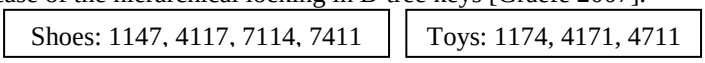

Figure 9. Records in a non-unique non-clustered index.

Figure 9 shows an example with two distinct key values and seven record identifiers. The key values are sorted in a B-tree; moreover, each list of record identifiers is also sorted. Key range locking can be applied both the key values (e.g,. “shoes”) and to pairs of key value and record identifier (e.g., “Toys 4711”). A lock might thus protect all records with a key value (e.g., “shoes”), the gap between two key values (e.g., between “shoes” and “toys”), a pair of key value and record identifier (e.g., “shoes 4117”), or a gap between two record identifiers (e.g., between “toys 4171” and “toys 4711”). There are really two hierarchies here: the hierarchy of key ranges and a hierarchy of key value, gap, and their combination. Each of these lock levels may be optimal depending on the operation (e.g., “insertion” or “select”) and the level of lock contention in the database.

The choice whether to lock unique values (and their clusters of row identifiers) or each pair of value and row identifier is independent of the choice of representation on disk or in memory. For example, a system may store each key value only once in order to save disk space yet lock individual pairs of key value and row identifier. Conversely, another system may store pairs of key value and record identifier yet lock all such pairs with the same value with a single lock, whether any form of compression is applied to such a redundant storage format or not.

Even in bitmap indexes, which employ particularly compressed representations for large sets of row identifiers, locks may protect row identifiers one at-a-time. The same is true for index formats that mix uncompressed bitmaps, compressed bitmaps, traditional lists of record identifiers, and lists of record identifiers compressed by storing distances between “on” bits in bitmaps. Note that bitmaps compressed with run-length encoding are quite similar to lists of record identifiers compressed by storing differences.

Another variety of bitmap indexes employs segmented lists, i.e., the domain represented by the bitmap is divided into segments and a bitmap is stored for each non-empty segment. For example, if a record identifier consists of a device identifier, page identifier, and slot number, then the device identifier plus some bytes of the page identifier may be interpreted as a segment identifier. Bitmaps then capture the remaining bytes of the page identifier and the slot number. The search key in the B-tree appends the segment identifier to the search key desired by the user. With respect to concurrency control, a single lock may cover all segments for a search key value or a separate lock may be required for each segment. In the latter case, insertion and deletion of empty segments is quite similar to record insertion and deletion in the simplest form of B-trees, both with respect to concurrency control and recovery.

All these variations on locking in non-unique indexes are special cases of the general orthogonality between the representation (including compression) of data in a data structure and the required concurrency control among user transactions (including locking), which is due, of course, to the separation of latches to protect data structures and locks to protect the logical database contents.

5.5 Increment lock modes

In addition to the traditional read and write locks, or shared and exclusive locks, other lock modes have been investigated. Most notable among those is the “increment” lock. Increment locks enable concurrent transactions to increment and decrement sums and counts. This is rarely required in detail tables but can be a concurrency bottleneck in materialized and indexed summary views, also known as “group by” views.

Increment locks are compatible with each other, but they are not compatible with read or write locks. Moreover, increment locks do not imply read permission for the lock holder. In other words, a transaction holding an increment lock cannot determine a current value without also acquiring a read lock. The combination of read lock and increment lock is equivalent to a write lock, because it excludes all other transactions from holding any kind of lock at the same time.

	S	*X*	*E*
S	ok		
X			
E			ok

Figure 10. Lock compatibility with increment locks.

Figure 10 shows the compatibility matrix. The increment lock is indicated with the latter E in reference to O'Neil's escrow locks [O'Neil 1986], although they have more semantics than merely incrementing a sum or count. Here, we assume that any addition or subtraction is permitted, with no concern for valid value ranges. The original escrow locks have such provisions, e.g., to prevent a negative account balance.

Increment locks in B-tree indexes are particularly useful for materialized and indexed summary views. Exclusive locks may be sufficient for updates in the base tables but, due to fewer rows in the summary view, may lead to unacceptable contention during view maintenance. Very much like reference counts in traditional memory management, records in such views must include a count field. This field serves as a generalization of the ghost bit; when the count is zero, the record is a ghost record not to be included in query results. Holding an increment lock, a user transaction may increment counts and sums thus turning a ghost record into a valid summary row and vice versa.

In grouped summary views, ghost records enable efficient creation and removal of groups [Graefe and Zwilling 2004]. Upon deletion of the last relevant row in the detail table, the reference count in the corresponding summary row reaches zero and thus turns the record into a ghost. Ghost clean-up can be deferred past end-of-transaction, e.g., until the space is required. When a new group must be created due to a new detail item with a unique group key, a ghost record can be created outside of the user transaction and can turn into a valid record when the user transaction commits an increment in the reference count. Note that no lock is required to insert or remove a new ghost record and key, only a latch for the data structure. Luo et al. [2005] developed an alternative solution for these issues based on special "value-based" latches.

Increment locks on key values immediately permit incrementing counts and sums in the record whose key is locked. Increment locks on large key ranges containing many keys and records permit increment operations in all records in the key range. Increment locks for an open interval between two neighboring key values can be defined, with perhaps surprising semantics and effects. They apply to any new ghost record inserted into the key range, i.e., the transaction starting with an increment lock on an open interval ends up holding increment locks on the newly inserted key as well as the two resulting open intervals. Based on the increment lock, the transaction may turn the ghost into a valid record by incrementing its counts and sums. Thus, an increment lock on an open interval permits insertion of new summary records and it prevents all other transactions from obtaining a shared or exclusive lock on the interval.

	NS	*NX*	*NE*	*SN*	*S*	*SX*	*SE*	*XN*	*XS*	*X*	*XE*	*EN*	*ES*	*EX*	*E*
NS	ok			ok	ok			ok	ok			ok	ok		
NX				ok				ok				ok			
NE			ok	ok			ok	ok			ok	ok			ok
SN	ok	ok	ok	ok	ok	ok	ok								
S	ok			ok	ok										
SX				ok											
SE			ok	ok			ok								
XN	ok	ok	ok												
XS	ok														
X															
XE			ok												
EN	ok	ok	ok									ok	ok	ok	ok
ES	ok											ok	ok		
EX												ok			
E			ok									ok			ok

Figure 11. Lock table key range locking and increment locks.

Figure 11 shows the lock compatibility matrix for key range locking with increment locks. The matrix is becoming a bit large, but it can be derived from Figure 10 just like Figure 8 from Figure 7. The derivation could be done by software either during development or start-up of the database management code. Alternatively, the large lock matrix of Figure 11 may not be stored at all and lock requests test two compatibilities as appropriate. The savings due to combined lock modes are in the number of lock manager invocations including latching and searching the lock manager's hash table; once a lockable resource is found in the hash table, two tests instead of one are a minor overhead.

This latter argument becomes even stronger if U lock modes are considered. As pointed out by Korth [1983], there can be many "upgrade" locks from one mode to another. For example, when some set of transactions holds a record in increment mode, some transaction might request an upgrade to a shared lock. In the traditional system based on only shared and exclusive locks, only one upgrade lock mode is useful and is commonly called the "update" lock; if additional basic lock modes are introduced such as the increment lock, the name "update" lock is no longer helpful or appropriate.

5.6 Summary

In summary, key range locking is the technique of choice for concurrency control among transactions. It is well understood, enables high concurrency, permits ghost records to minimize effort and "lock footprint" in user transactions, adapts to any key type and key distribution, and prevents phantoms for true serializability. By strict separation of locks and latches, of abstract database contents and in-memory data structures including cached database representation, and of transactions and threads, previously difficult techniques become clear. This clarity enables efficient implementations, correct tuning, and innovation with advanced lock modes and scopes, exemplified by increment locks and key range locking for separator keys in upper B-tree nodes.

6 Future directions

Perhaps the most urgently needed future direction is simplification. Functionality and code for concurrency control and recovery are too complex to design, implement, test, debug, tune, explain, and maintain. Elimination of special cases without a severe drop in performance or scalability would be welcome to all database development and test teams.

At the same time, B-trees and variants of B-tree structures are employed in new areas, e.g., Z-order UB-trees for spatial and temporal information, various indexes for unstructured data and XML documents, in-memory and on-disk indexes for data streams and as caches of reusable intermediate query results. It is unclear whether these application areas require new concepts or techniques in B-tree concurrency control.

Some implementation techniques need to be revised for optimal use of modern many-core processors. For example, will traditional B-tree primitives be divided into concurrent operations? Could the operations discussed earlier for B^{link}-trees and for ghost records serve as the basis for such highly concurrent primitives? Will those processors offer new hardware support for concurrency control, to be used for critical sections instead of latches? Is transactional memory a suitable replacement for latches? Can the implementation of database management systems benefit from hardware transactions with automatic rollback and restart? Depending on the answers to these questions, high concurrency in B-tree operations may become possible with fairly simple and compact code.

7 Summary and conclusions

In summary, concurrency control for B-tree indexes in databases can be separated into two levels: concurrent threads accessing in-memory data structures and concurrent transactions accessing database contents. These two levels are implemented with latches and locks.

Latches support a limited set of modes such as shared and exclusive, they do not provide advanced services such as deadlock detection or escalation, and they can often be embedded in the data structures they protect. Therefore, their acquisition and release can be very fast, which is important as they implement short critical sections in the database system code.

Locks support many more modes than latches and provide multiple advanced services. Management of locks is separate from the protected information, for example, keys and gaps between keys in the leaf page

of a B-tree index. The hash table in the lock manager is in fact protected itself by latches such that many threads can inspect or modify the lock table as appropriate.

The principal technique for concurrency control among transactions accessing B-tree contents is key range locking. Various forms of key range locking have been designed. The most recent design permits separate locks on individual key values and on the gaps between key values, applies strict multi-granularity locking to each pair of a key and a neighboring gap, reduces lock manager invocations by using additional lock modes that can be derived automatically, enables increment locks in grouped summary views, and exploits ghost records not only for deletions for but also for insertions.

Acknowledgements

Helpful feedback and encouragement by Theo Härder, Harumi Kuno, Gary Smith, the anonymous ACM TODS reviewers, and last not least the associate editor Hank Korth are gratefully acknowledged.

References

Bayer, R., 1997: The universal B-Tree for multidimensional indexing: general concepts. WWCA: 198-209.

Bayer, B., McCreight, E. M., 1972: Organization and maintenance of large ordered indices. Acta Inf. 1: 173-189.

Bayer, R., Schkolnick, M., 1977: Concurrency of operations on B-trees. Acta Inf. 9: 1-21.

Bayer, R., Unterauer, K., 1977: Prefix B-trees. ACM TODS 2(1): 11-26.

Bernstein, P. A., Hadzilacos, V., Goodman, N., 1987: Concurrency control and recovery in database systems. Addison-Wesley.

Comer, D., 1979: The ubiquitous B-tree. ACM Comput. Surv. 11(2): 121-137.

Eswaran, K. P., Gray, J., Lorie, R. A., Traiger I. L., 1976: The notions of consistency and predicate locks in a database system. Comm. ACM 19(11): 624-633.

Gawlick, D., Kinkade, D., 1985: Varieties of concurrency control in IMS/VS Fast Path. IEEE Database Eng. Bull. 8(2): 3-10.

Graefe, G., 2007: Hierarchical locking in B-tree indexes. BTW Conf.: 18-42.

Graefe, G., Zwilling, M. J., 2004: Transaction support for indexed views. ACM SIGMOD: 323-334.

Gray, J., Reuter, A., 1993: Transaction processing: concepts and techniques. Morgan Kaufmann.

Jaluta, I., Sippu, S., Soisalon-Soininen, E., 2005: Concurrency control and recovery for balanced B-link trees. VLDB J. 14(2): 257-277.

Jordan, J. R., Banerjee, J., Batman, R. B., 1981: Precision locks. ACM SIGMOD: 143-147.

Korth, H. F., 1983: Locking primitives in a database system. J. ACM 30(1): 55-79.

Kung, H. T., Robinson, J. T., 1981: On optimistic methods for concurrency control. ACM TODS 6(2): 213-226.

Larus, J. R., Rajwar, R., 2006: Transactional memory. Synthesis lectures on computer architecture. Morgan & Claypool Publishers.

Lehman, P. L., Yao, S. B., 1981: Efficient locking for concurrent operations on B-trees. ACM TODS 6(4): 650-670.

Lomet, D. B., 1993: Key range locking strategies for improved concurrency. VLDB: 655-664.

Lomet, D. B., 2004: Simple, robust and highly concurrent B-trees with node deletion. ICDE: 18-28.

Luo, G., Naughton, J. F., Ellmann, C. J., Watzke, M., 2005: Locking protocols for materialized aggregate join views. IEEE TKDE 17(6): 796-807.

Mohan, C., 1990: ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. VLDB: 392-405.

Mohan, C., Levine, F., 1992: ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD: 371-380.

Moss, J. E. B., 2006: Open nested transactions: semantics and support. Workshop on memory performance issues (WMPI), Austin, TX.

Ni, Y., Menon, V., Adl-Tabatabai, A.-R., Hosking, A. L., Hudson, R. L., Moss, J. E. B., Saha, B., Shpeisman, T., 2007: Open nesting in software transactional memory. PPoPP: 68-78.

O'Neil, P. E., 1986: The Escrow transactional method. ACM TODS 11(4): 405-430.

Ramsak, F., Markl, V., Fenk, R., Zirkel, M., Elhardt, K., Bayer, R., 2000: Integrating the UB-tree into a database system kernel. VLDB: 263-272.

Srinivasan, V., Carey, M. J., 1991: Performance of B-tree concurrency algorithms. ACM SIGMOD: 416-425.

Weikum, G., 1991: Principles and realization strategies of multilevel transaction management. ACM TODS 16(1): 132-180.

Weikum, G., Schek, H.-J., 1992: Concepts and applications of multilevel transactions and open nested transactions. Database transaction models for advanced applications: 515-553.

Weikum, G., Vossen, G., 2002: Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Morgan Kaufmann.

Zou, C., Salzberg, B., 1996: On-line reorganization of sparsely-populated B^+-trees. ACM SIGMOD: 115-124.