category, and a bubble attached to a node represents a set of archaeological records. The size of a bubble attached to a node reflects the number of records belonging to that category. The hyperbolic tree supports "focus + context" navigation; it also provides an overview of records organized in the archaeological digital library. It shows that the records are from seven archaeological sites (the Megiddo site has the most) and are of twelve different types.

Future Directions

Browsing and searching are often provided by digital libraries as separate services. Developers commonly see these functions as having different underlying mechanisms, and they follow a functional, rather than a task-oriented, approach to interaction design. While exhibiting complementary advantages, neither paradigm alone is adequate for complex information needs. Searching is popular because of its ability to identify information quickly. On the other hand, browsing is useful when appropriate search keywords are unknown or unavailable to users. Browsing also is appropriate when a great deal of contextual information is obtained along the navigation path. Therefore, a synergy between searching and browsing is required to support users' information seeking goals. Browsing and searching can be converted and switched to each other under certain conditions [5]. This suggests some new possibilities for blurring the dividing line between browsing and searching. If these two services are not considered to have different underlying mechanisms, they will not be provided as separated functions in digital libraries, and may be better integrated.

Text mining and visualization techniques provide digital libraries additional powerful exploring services, with possible beneficial effects on browsing and searching. Digital library exploring services such as browsing, searching, clustering, and visualization can be generalized in the context of a formal digital library framework [5]. The theoretical approach may provide a systematic and functional method to design and implement exploring services for domain focused digital libraries.

Experimental Results

See Fig. 2 – Fig. 6 above, and the corresponding explanation.

Data Sets

See Fig. 2 – Fig. 6 above, and the ETANA Digital Library [5].

Cross-references

▶ Digital Libraries

Recommended Reading

1. Baeza-Yates R. and Ribeiro-Neto B. Modern Information Retrieval. Addison-Wesley, Reading, MA, 1999.
2. Fox E.A. and Urs S.R. Digital libraries, chap. 12. In Annual Review of Information Science and Technology, Vol. 36, B. Cronin (ed.); Medford, NJ, Information Today, Inc. 2002, pp. 503–589.
3. Fox E.A., Rous B., and Marchionini G. ACM's hypertext and hypermedia publishing projects. In Hypertext/Hypermedia Handbook, E. Berk, J. Devlin (eds.). McGraw-Hill, NY, 1991, pp. 465–467.
4. Goncalves M., Fox E.A., Watson L., and Kipp N. Streams, structures, spaces, scenarios, societies (5S): a formal model for digital libraries. ACM Trans. Inf. Syst., 22(2):270–312, 2004.
5. Shen R., Vemuri N., Fan W., Torres R., and Fox E.A. Exploring digital libraries: integrating browsing, searching, and visualization. In Proc. 6th ACM/IEEE-CS joint Conference on Digital Libraries, 2006, pp. 1–10.

B-Tree

▶ B+-Tree

B-Tree Concurrency Control

▶ B-Tree Locking

B-Tree Locking

Goetz Graefe
Hewlett-Packard Laboratories, Palo Alto, CA, USA

Synonyms

B-tree concurrency control; Row-level locking; Key value locking; Key range locking; Lock coupling; Latching; Latch coupling; Crabbing

Definition

B-tree locking controls concurrent searches and updates in B-trees. It separates transactions in order to protect the B-tree contents and it separates threads in order to protect the B-tree data structure. Nowadays, the latter is usually called latching rather than locking.

Historical Background

Bayer and Schkolnick [1] presented multiple locking (latching) protocols for B^*-trees (all data records in the leaves, merely separator keys 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 reserving appropriate locks (latches) for ancestors of unsafe nodes.

Lehman and Yao defined B^{link}-trees by relaxing the B-tree structure in favor of higher concurrency [8]. Srinivasan and Carey demonstrated their high performance using detailed simulations [13]. Jaluta et al. recently presented a detailed design for latching in B^{link}-trees, including a technique to avoid excessive link chains and thus poor search performance [7].

IBM's System R project explored multiple transaction management techniques, including transaction isolation levels and lock duration, predicate locking and key value 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 [14] and into open nested transactions [3,12] 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 [10,11] explicitly separates locks and latches, 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 separate 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.

Lomet's design for key range locking [4] 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 [9] applies traditional hierarchical locking to keys and gaps (open intervals) between keys, employs ghost (pseudo-deleted) records during insertion as well as during deletion, and permits more concurrency with fewer special cases. The same paper also outlines hierarchical locking exploiting B-trees' hierarchical structure or multi-field B-tree keys.

Foundations

The foundations of B-tree locking are the well-known transaction concepts, including multi-level transactions and open nested transactions, and pessimistic concurrency control, i.e., locking. Multiple locking concepts and techniques are employed, including two-phase locking, phantom protection, predicate locks, precision locks, key value locking, key range locking, multi-granularity locking, hierarchical locking, and intention locks.

Preliminaries

Most work on concurrency control and recovery in B-trees assumes what Bayer and Schkolnick call B^*-trees [1] and what Comer calls B^+-trees [2], 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. Following this tradition, this entry ignores the original design of B-trees with data records in interior nodes.

Also ignored are 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. Among the ignored techniques are 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, 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.

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 should not observe intermediate and incomplete data structures. The difference also becomes apparent in the opposite case when a single operating system thread is multiplexed to serve all user transactions.

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, which can be confusing.

Locks separate transactions using read and write locks on pages, on B-tree keys, or even 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 for 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.

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.

Latch Coupling and B^{link}-Trees

Latches coordinate multiple concurrent threads accessing shared in-memory data structures, including images of on-disk storage structures while in the buffer pool. In the context of B-trees, latches solve several problems that are similar to each other but nonetheless lend themselves to different solutions.

First, a page image in the buffer pool must not be modified (written) by one thread while it is interpreted (read) by another thread. For this issue, 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. Latches are useful not only for pages in the buffer pool but also for the buffer pool's table of contents or the lock manager's hash table.

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 retaining the latch on the parent node until the child node is latched. This technique is traditionally called "lock coupling" or better "latch coupling."

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 scan. This issue is similar to the previous, with two differences. On the positive side, asynchronous read-ahead may alleviate the frequency of buffer faults. On the negative side, deadlock avoidance

among scans in opposite directions requires that latch acquisition code provides an immediate failure mode.

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 root node splits and a new root node is added. Going back from the leaf towards the B-tree root works well in single-threaded B-tree implementations, but in multi-threaded code it introduces the danger of deadlocks. This issue 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.

One approach latches all nodes in exclusive mode during the root-to-leaf traversal. The obvious problem in this approach is the potential concurrency bottleneck, particularly at a B-tree's root. Another approach performs the root-to-leaf search using shared latches and attempts an upgrade to an exclusive latch when necessary. A third approach reserves nodes using "update" or "upgrade" latches. A refinement of these three 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.

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 a latch) of the second approach. Its disadvantage is that it wastes some space by splitting earlier than truly required. 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.

An entirely different approach relaxes the data structure constraints of B-tress and divides a node split into two independent steps. Each node has a high fence key and a pointer to its right neighbor, thus the name B^{link}-trees. The right neighbor might not yet be referenced in the node's parent and a root-to-leaf search might need to proceed to the node's right neighbor. 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 should happen as soon as possible yet it may be delayed beyond a system reboot or even a crash. 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.

Key Range Locking

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. High concurrency requires a fine granularity of locking, e.g., locking individual keys in B-tree indexes. The terms key value locking and key range locking are often used interchangeably.

Key range locking is a special form of predicate 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, which form half-open intervals including the gap between two neighboring keys and one of the end points.

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.

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 $+\infty$ 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.

In those B-tree implementations in which a deletion does not actually erase the record and instead merely marks the record as invalid, "pseudo-deleted," or a "ghost" record, each ghost record's key participates in

key range locking just like a valid record's key. Another technique to increase concurrency models a key, the appropriate neighboring open interval, and the combination of key and open interval as three separate items [9]. These items form a hierarchy amenable to multi-granularity locking. Moreover, since key, open interval, and their combination are all identified by the key value, additional lock modes can replace multiple invocations of the lock manager by a single one, thus eliminating the execution costs of this hierarchy.

Multi-granularity locking also applies keys and individual rows in a non-unique index, whether such rows are represented using multiple copies of the key, a list of row identifiers associated with a single copy of the key, or even a bitmap. Multi-granularity locking techniques exploiting a B-tree's tree structure or a B-tree's compound (multi-column) key have also been proposed. Finally, "increment" locks may be very beneficial for B-tree indexes on materialized summary views [5].

Both proposals need many details worked out, e.g., appropriate organization of the lock manager's hash table to ensure efficient search for conflicting locks and adaptation during structure changes in the B-tree (node splits, load balancing among neighboring nodes, etc.).

Key Applications

B-tree indexes have been called ubiquitous more than a quarter of a century ago [2], and they have become ever more ubiquitous since. Even for single-threaded applications, concurrent threads for maintenance and tuning require concurrency control in B-tree indexes, not to mention online utilities such as online backup. The applications of B-trees and B-tree locking are simply too numerous to enumerate them.

Future Directions

Perhaps the most urgently needed future direction is simplification – concurrency control and recovery functionality and code are too complex to design, implement, test, tune, explain, and maintain. Elimination of any 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 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.

Online operations – load and query, incremental online index creation, reorganization & optimization, consistency check, trickle load and zero latency in data warehousing including specialized B-tree structures.

Scalability – granularities of locking between page and index based on compound keys or on B-tree structure; shared scans and sort-based operations including "group by," merge join, and poor man's merge join (index nested loops join); delegate locking (e.g., locks on orders cover order details) including hierarchical delegate locking.

B-tree underpinnings for non-traditional database indexes, e.g., blobs, column stores, bitmap indexes, and master-detail clustering.

Confusion about transaction isolation levels in plans with multiple tables, indexes, materialized and indexed views, replicas, etc.

URL to Code

Gray and Reuter's book [6] shows various examples of sample code. In addition, the source of various open-source database systems is readily available.

Cross-references

- ▶ Concurrency Control and Recovery
- ▶ Database benchmarks – online transaction processing
- ▶ Locking Granularity and Lock Types
- ▶ pessimistic concurrency control
- ▶ phantoms
- ▶ precision locks
- ▶ predicate locks
- ▶ relational data warehousing
- ▶ System Recovery
- ▶ two-phase commit
- ▶ two-phase locking
- ▶ write-ahead logging

Recommended Reading

1. Bayer R. and Schkolnick M. Concurrency of operations on B-trees. Acta Inf., 9:1–21, 1977.
2. Comer D. The ubiquitous B-tree. ACM Comput. Surv., 11(2):121–137, 1979.
3. Eliot J. and Moss B. Open Nested Transactions: Semantics and Support. In Proc. Workshop on Memory Performance Issues 2006.

4. Graefe G. Hierarchical locking in B-tree indexes. BTW Conf., 2007, pp. 18–42.
5. Graefe G. and Zwilling M.J. Transaction support for indexed views. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004.
6. Gray J. and Reuter A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, San Francisco, CA, 1993.
7. Jaluta I., Sippu S., and Soisalon-Soininen E. Concurrency control and recovery for balanced B-link trees. VLDB J., 14(2):257–277, 2005.
8. Lehman P.L. and Yao S.B. Efficient locking for concurrent operations on B-trees. ACM Trans. Database Syst., 6(4):650–670, 1981.
9. Lomet D.B. Key range locking strategies for improved concurrency. In Proc. 19th Int. Conf. on Very Large Data Bases, 1993, pp. 655–664.
10. Mohan C. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In Proc. 16th Int. Conf. on Very Large Data Bases, 1990, pp. 392–405.
11. Mohan C., Haderle D.J., Lindsay B.G., Pirahesh H., and Schwarz P.M. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM Trans. Database Syst., 17(1):94–162, 1992.
12. Ni Y., Menon V., Adl-Tabatabai A-R., Hosking AL., Hudson RL., Moss JEB., Saha B., and Shpeisman T. Open nesting in software transactional memory. In Proc. 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming, 2007, pp. 68–78.
13. Srinivasan V. and Carey M.J. Performance of B-tree concurrency algorithms. In Proc. ACM SIGMOD Int. Conf. on Management of Data, 1991, pp. 416–425.
14. Weikum G. Principles and realization strategies of multilevel transaction management. ACM Trans. Database Syst., 16(1):132–180, 1991.

Buffer Management

Giovanni Maria Sacco
University of Torino, Torino, Italy

Definition

The database buffer is a main-memory area used to cache database pages. Database processes request pages from the buffer manager, whose responsibility is to minimize the number of secondary memory accesses by keeping needed pages in the buffer. Because typical database workloads are I/O-bound, the effectiveness of buffer management is critical for system performance.

Historical Background

Buffer management was initially introduced in the 1970s, following the results in virtual memory systems. One of the first systems to implement it was IBM System-R. The high cost of main-memory in the early days forced the use of very small buffers, and consequently moderate performance improvements.

Foundations

The buffer is a main-memory area subdivided into frames, and each frame can contain a page from a secondary storage database file. Database pages are requested from the buffer manager. If the requested page is in the buffer, it is immediately returned to the requesting process with no secondary memory access. Otherwise, a fault occurs and the page is read into a free frame. If no free frames are available, a "victim" page is selected and its frame is freed by clearing its content, after writing it to secondary storage if the page was modified. Usually any page can be selected as a victim, but some systems allow processes actively using a page to fix or pin it, in order to prevent the buffer manager from discarding it [5]. Asynchronous buffered write operations have an impact on the recovery subsystem and require specific protocols not discussed here.

There are obvious similarities between buffer management and virtual memory (VM) systems [3]. In both cases, the caching system tries to keep needed pages in main-memory in order to minimize secondary memory accesses and hence speed up execution. As in VM systems, buffer management is characterized by two policies: the *admission policy*, which determines when pages are loaded into the buffer, and the *replacement policy*, which selects the page to be replaced when no empty frames are available. The admission policy normally used is demand paging (i.e., a missing page is read into the buffer when requested by a process), although prefetching (pages are read before processes request them) was studied (e.g., [1]). Since the interaction with the caching system is orders of magnitude less frequent in database systems than in VM systems, "intelligent" replacement policies such as LRU [3] (the Least Recently Used page is selected for replacement) can be implemented in software, with no performance degradation. Inverted page tables are used because their space requirement is proportional to the buffer size rather than to the entire database space as in