Foster B-trees – concepts and state transitions

Goetz Graefe
Hewlett-Packard Laboratories

Abstract

Foster B-trees are a new variant of the B-tree data structure. B-tree indexes are well-known and variants of B-trees are used ubiquitously in databases, key-value stores, information retrieval, and file systems. Whereas the original design of B-trees was optimized for disk storage 40 years ago, it is not optimal for modern hardware. The purpose of this new design is to combine advantages of prior B-tree variants optimized for many-core processors, storage arrays, flash storage, and non-volatile memory. The specific goals are

i. minimal concurrency control requirements for the data structure,
ii. efficient migration of nodes to new storage locations, and
iii. support for continuous and comprehensive self-testing.

The concurrency optimizations pertain to the physical data structure. They are orthogonal to concurrency control for the logical contents. In database terms, Foster B-trees optimize latching without imposing restrictions or specific designs on transactional locking, e.g., key range locking.

Efficient migration of pages permits writing pages in contiguous groups, e.g., in entire stripes of disk arrays or entire erasure blocks of flash storage. For file systems, this is known as log-structured file system [Rosenblum and Ousterhout 1992]; in database systems, this is known as write-optimized B-trees [Graefe 2004]. Both can be exploited not only for large writes but also for wear leveling. In both contexts, page migration is invoked when a dirty page in the buffer pool is written to permanent storage. In addition, efficient page migration permits efficient defragmentation of B-trees in file systems and in databases.

The inexpensive yet continuous and comprehensive verification of all invariants goes beyond error-correcting codes within each storage page; it includes all cross-node invariants of the B-tree structure. Whether inconsistencies are introduced by mistakes in the B-tree code, in lower software layers such as replication or a file system, or in the hardware (including endurance problems), violations of B-tree invariants can be detected reliably and efficiently. Efficient recovery of individual pages is enabled by recent complementary work that is independent of any specific data structure such as Foster B-trees.

1 Introduction

The design goal for Foster B-trees has been to combine the advantages of B^{link}-trees, symmetric fence keys, and write-optimized B-trees. Some of their essential characteristics contradict each other: while B^{link}-trees absolutely require left-to-right sibling pointers, write-optimized B-trees enable efficient page migration precisely by avoiding sibling pointers. In addition to resolving this contradiction, Foster B-trees enable efficient node deletion as well as load balancing among sibling nodes.

1.1 B^{link}-trees

Concurrency control in B-tree indexes is divided into locking, which coordinates concurrent transactions and protects logical database contents, and latching, which coordinates concurrent threads and protects in-memory data structures including images of disk pages [Graefe 2010]. The following discussion is about latching, i.e., protection of data structures. The techniques and discussions here are orthogonal to key-value locking and related transactional techniques.

Bayer and Schkolnick [1977] proposed a latching scheme for root-to-leaf passes in B-trees. That technique, centered on the notion of a safe node, has root-to-leaf search passes retain page latches until they cannot possibly be needed. For example, the root-to-leaf pass for an insertion retains an exclusive latch until it passes through a node that cannot require a split because there is sufficient free space for a local insertion in case a child node must split and post a new branch key (also known as separator key or guide key) in the parent node. – This scheme suffered not only from complexity and unpredictable latch retention times but also from a weak definition of safe nodes in B-tree indexes with variable-size key values.

Lehman and Yao [1981] proposed B^{link}-trees as a remedy for the problems of the prior latching (locking) system described for B-tree structures. This design relies on left-to-right sibling pointers in addition to parent-to-child pointers. Each level of the B-tree data structure forms a singly-linked list. When a node overflows and a new node is required, the overflowing node holds a pointer to the new node together with a key value that separates key values retained in the old node and those moved to the new node. Soon thereafter, the pointer and the branch key are copied to the parent node. If thereupon the parent node overflows, the same techniques is applied there.

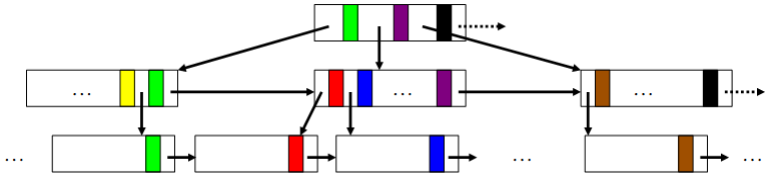

Figure 1. A B^{link}-tree.

Figure 1 shows a B^{link}-tree. (In this paper, each diagram or sequence of diagrams indicates equal key values by equal colors. White and black represent $-\infty$ and $+\infty$, respectively, i.e., values outside the user's key domain. Key values immaterial for the discussion at hand are omitted. Boxes are sized for convenience of illustration. B-tree nodes of varying sizes are not considered in this paper.) The diagram shows, in particular, the parent-to-child pointers and the left-to-right-sibling pointers. In other words, each node has two incoming pointers in the steady state of B^{link}-trees. There are *nil* pointers along the right edge of the B^{link}-tree.

The innovative advantage of B^{link}-trees is that they require only two page latches at a time, even during structural modifications of the B-tree. For example, when a node is first split, two latches protect the overflowing node and the new node, whereas the parent node does not participate in the split operation. When pointer and branch key are copied from the formerly overflowing node to the parent node, only those two nodes are latched, whereas the new node does not participate in the copy operation.

Various variants put different constraints on the basic B^{link}-tree. For example, the original design requires that the split operation must post the branch key in the parent before it may complete, whereas other designs leave the copy operation to a subsequent root-to-leaf pass, possibly within another thread executing another transaction. Similarly, some designs prohibit splitting the formerly overflowing node or the new node until branch key and pointer are posted in the parent, whereas other designs permit chains to form. Prohibiting chains guarantees a logarithmic number of nodes on any root-to-leaf pass, whereas short-lived chains with reasonably aggressive chain dissolution by posting branch keys and child pointers seems more tolerant of local spikes of contention without significant performance impact.

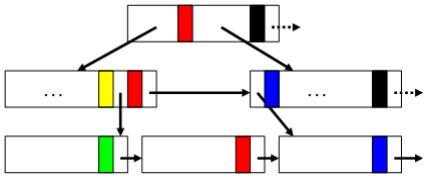

Figure 2. Intermediate state during node insertion in a B^{link}-tree.

The core innovation of B^{link}-trees is the division of node insertion into two steps and the intermediate state during node insertion. The initial step merely creates and fills an overflow node for the overflowing node; the final step repairs the B-tree by copying branch key and node pointer into the parent node and thus guarantees future root-to-leaf passes with a logarithmic number of nodes. Figure 2 illustrates the intermediate step while dividing the overflowing (left leaf) node.

In other words, the initial step puts a new branch key and a pointer value into the overflowing node; the final step copies the branch key and the pointer value to the parent node. Thus, in stable state, each node's address and lowest possible key value are stored in two other nodes, i.e., in the parent node and in the left sibling node.

The intermediate state defined for node insertion could also be used for other operations, e.g., during node deletion and for load balancing among sibling nodes, but those operations would require more than two latches at a time. An alternative design for node deletion [Lomet 2004] relies on a global latch (one per B-

tree), in scope comparable to the "tree modification latch" in the ARIES designs for B-trees [Mohan 1990, Mohan and Levine 1992]. B^{link}-trees have not been used (much) in real implementations.

1.2 Symmetric fence keys

A B-tree index may become inconsistent due to software or hardware failures. For example, Figure 3 (copied from [Graefe and Stonecipher 2009] together with this explanation) shows the result of incorrectly splitting a leaf node in a B-tree with forward and backward pointers among siblings. When leaf node b was split and leaf node c was created, the backward pointer in successor node d incorrectly remained unchanged. A subsequent (descending) scan of the leaf level will produce a wrong query result, and subsequent split and merge operations will create further havoc.

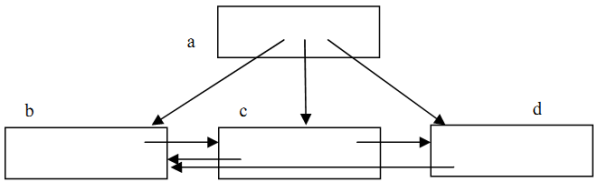

Figure 3. An incomplete leaf split.

The problem might arise after an incomplete execution, an incomplete recovery, or an incomplete replication of the split operation. The cause might be a defect in the database software, e.g., in the buffer pool management, or in the storage management software, e.g., in snapshot or version management. In other words, there are many thousands of lines of code that may contain a defect that leads to a situation like the one illustrated in Figure 3.

Different B-tree inconsistencies result if all nodes but c are saved correctly in the database; in that case, a, b, and d point to a page full of garbage. If all nodes but b are saved correctly, the key range in b would not conform to the separator keys in a. If all nodes but a are saved correctly, a search for records in node c will not find them. B-tree verification must find all such errors reliably and as efficiently as possible. Of course, efficient verification also reduces the software development cycle for the B-tree code because it permits efficient testing with frequent verification during development and tuning of the B-tree code.

Comprehensive B-tree verification must verify all invariants within each node (page) and all cross-node invariants about all key values and all pointers. This is fairly straightforward in an offline index-order scan, but it is also quite inefficient and disruptive. Some variants of B-trees permit offline disk-order scans, even parallel scans, i.e., as the verification logic can process B-tree nodes in any order. Some implementations permit online verification using snapshot logic in the file system; the problem with this approach is that the verification result is already out of date when the verification completes. Usually, the hardest problem is verification of cousin nodes, i.e., siblings that do not share a parent node but do share a grandparent or even higher ancestor node.

For comprehensive online consistency checks, both low and high boundary values are required, called fence keys from here on. These fence keys are precise copies of branch keys in ancestor nodes. For offline consistency checks, symmetric fence keys (or low and high fence keys) enable efficient comprehensive structural verification of B-trees even if pages and nodes are read in an unpredictable order, e.g., from a parallel scan or during a parallel backup operation. For online consistency checks, all required and possible checking can easily be achieved as side effects of root-to-leaf B-tree searches, e.g., in database query execution. Each parent-to-child step can easily compare the branch keys associated with a child pointer with the fence keys in the child page.

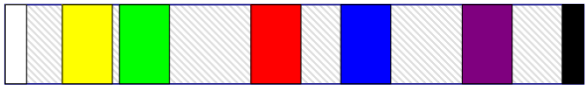
Figure 4. A B-tree node with symmetric fence keys.

Figure 4 illustrates a page with some full-size records as well as two short fence keys. In general, one of the fence keys is an inclusive bound and the other is an exclusive bound. The inclusive bound might be a valid data record or a ghost record (also known as pseudo-deleted record); the exclusive bound cannot be a

valid record. In Figure 4, the fence keys have the values −∞ and +∞ (white and black), which indicates that this node is a root node.

As observed by Lomet [2001], symmetric fence keys aid in page-wide prefix truncation [Bayer and Unterauer 1977]. They also simplify key range locking in some cases, namely when branch keys guide a root-to-leaf search to one node but the key value that needs to be locked for phantom protection, i.e., serializable transaction isolation [Gray et al. 1976], is located in a sibling node. If fence key values participate in key range locking, this case cannot occur.

1.3 Write-optimized B-trees

The original design point of write-optimized B-trees [Graefe 2004] was enabling large writes in RAID-6 arrays with even higher small-write penalty than RAID-4/5 arrays [Chen et al. 1994]. However, write-optimized B-trees are equally useful with flash storage, where they can enable both large writes (as large as entire erase blocks) and wear leveling (by inexpensive page migration) without a flash translation layer, i.e., without another level of indirection with lookup and maintenance overheads.

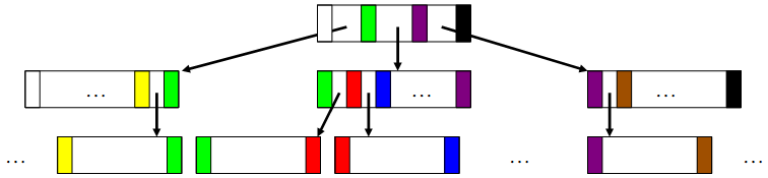

Figure 5. A write-optimized B-tree.

The essence of write-optimized B-trees is to avoid side pointers or sibling pointers, as shown in Figure 5. With only parent-to-child pointers remaining, moving a B-tree node to a new location requires an update of only one database page, namely the parent page. Page movement becomes inexpensive, both when assembling large write blocks and during defragmentation, should it become necessary. On the other hand, as B^{link}-trees rely on sibling pointers, the benefits of B^{link}-trees and of write-optimized B-trees seem contradictory.

The loss of one sibling pointer per node (compared to B^{link}-trees) or of two sibling pointers per node (compared to many industrial B-tree implementations) also implies a loss of redundant information useful for consistency checking and perhaps database repair. With modern hardware requiring ever-more fine-grained concurrency control, programming mistakes are a real possibility, however. Moreover, modern storage hardware with known reliability and endurance issues also suggest more rather than less consistency checking, and continuous online rather than occasional offline consistency checking. Thus, some redundancy between B-tree nodes is desirable but not in the form of sibling pointers.

1.4 Combined advantages

The combination of these goals and techniques, including efficient latching, efficient page migration, and efficient online consistency checks, motivates and defines Foster B-trees. In other words, each of the prior designs (B^{link}-trees, symmetric fence keys, and write-optimized B-trees) addressed one particular issue (simple latching with high concurrency, continuous yet comprehensive self-testing, and page migration for RAID and flash storage), but Foster B-trees address all three issues. The new design combines techniques from all three prior designs yet avoids their disadvantages and their contradictions. The disadvantages to be avoided include complex node deletion with global latches [Lomet 2004, Mohan 1990, Mohan and Levine 1992], lack of support for structural B-tree changes other than node insertion (e.g., load balancing among sibling nodes), and structural B-tree changes with more than two latches or with a leaf-to-root latch sequence.

2 Foster B-tree states and operations

A minimal Foster B-tree consists of a single node, a leaf node containing key values and the desired user-defined contents. An implementation may require that the root node is always a branch node, i.e., a node with branch keys and child pointers but not user-defined contents. In that case, the minimal tree consists of two nodes, a branch node and a leaf node. Prefix and suffix truncation [Bayer and Unterauer 1977] as well as other forms of compression are desirable but not necessary.

Like B[link]-trees, Foster B-trees split nodes locally without immediate upward propagation; therefore, only two latches at a time suffice. Like write-optimized B-trees [Graefe 2004], Foster B-trees permit only a single incoming pointer per node at all times; therefore, they support efficient page migration and defragmentation. Due to symmetric fence keys [Graefe and Stonecipher 2009], Foster B-trees permit continuous self-testing of all invariants; therefore, they enable very early detection of any page corruption.

Moreover, the restriction to a single incoming pointer at all times enables very simple and efficient node deletion. As during node insertion, two local latches also suffice during node deletion, load balancing, etc.

In the usual and most desirable state, a Foster B-tree looks precisely like a write-optimized B-tree as shown in Figure 5. The new aspect is in the sequence of intermediate states during structural changes, e.g., during split of an overflowing node and thus insertion of a new node.

2.1 Node splits in Foster B-trees

The intermediate state during a leaf split is transient and resolved quickly after it has been created, but it may persist long enough to be observed by other threads or other transactions. Resolving it means moving pointer and branch key from the formerly overflowing sibling node to the parent. In the following, we sometimes call the parent node the "permanent parent." The (formerly overflowing) sibling acts as temporary parent node and is called a "foster parent." (In the standard usage of the term, foster parents provide parental care and nurture to children not related through legal or blood ties.) We also use the terms "foster child" (a node with a foster parent), "foster relationship" (between foster parent and foster child), "permanent parent" (the standard parent in a B-tree), and "adoption" (ending a foster relationship).

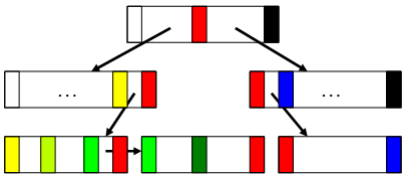

Figure 6. Intermediate state in a Foster B-tree.

Figure 6 illustrates the intermediate state during node insertion. The left leaf node was split and became a foster parent. In addition to its two fence keys, a foster parent also carries a third key, the foster key. This key value separates key values in the foster parent and in the foster child. In a foster parent, the high fence key is equal to a branch key in the parent and to the high fence key in the foster child, whereas the foster key is equal to the low fence key in the foster child. These equalities can easily be verified during root-to-leaf search in the B-tree for continuous yet comprehensive verification of all structural invariants.

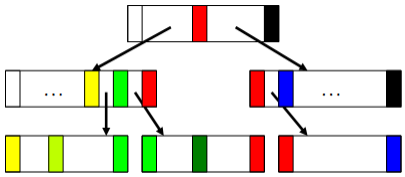

Figure 7. Final state after the node insertion.

Figure 7 shows the B-tree of Figure 6 after the newly allocated node has been adopted by the permanent parent, i.e., after the foster key and foster child pointer have been moved from the foster parent to the permanent parent. Note that in the former foster parent, the former foster key has become the high fence key.

Figure 6 and Figure 7 epitomize Foster B-trees: root-to-leaf passes of uniform length in the steady state, symmetric fence keys in every node, a single incoming pointer per node at all times, local overflow in the form of a foster relationship, load balancing between foster parent and foster child, and complex state transitions in multiple simple, transacted steps.

The essential difference to write-optimized B-trees is the intermediate state during an insertion (and other structural operations) with a pointer to a sibling node. In other words, write-optimized B-trees are strictly limited to pointers between nodes of adjacent tree levels, which guarantees a logarithmic tree depth at all times, whereas Foster B-trees permit temporary sibling pointers, which implies that a root-to-leaf pass may temporarily be longer than minimal.

The essential difference to B[link]-trees is that a foster relationship is resolved by moving the pointer and branch key, whereas they are copied in a B[link]-tree. In other words, Foster B-trees preserve the invariant that each node has only a single incoming pointer, even if structural B-tree modifications are divided into and carried out in multiple steps with intermediate states visible to other threads and other transactions.

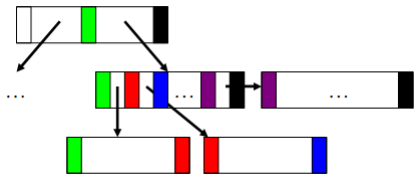

Figure 8. Branch node with foster child.

After splitting a leaf node, moving branch key and child pointer to the parent node may force an overflow there. If so, a new branch node is allocated, linked to the overflowing branch node as its foster child, and filled with some of the information from the overflowing foster parent. As soon as possible, the appropriate branch key and child pointer are moved to the parent's parent node, i.e., the grandparent node of the leaf node that had split. Figure 8 and Figure 9 illustrate the intermediate and final states. If required, ancestors up the tree map split, repeating the same steps at each level as required. Adding a new root node to a Foster B-tree is covered later.

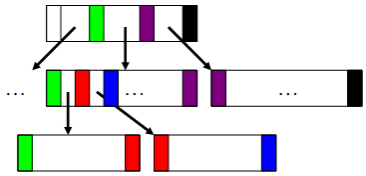

Figure 9. After adoption into the grandparent level.

If a foster parent or a foster child splits before their foster relationship is resolved by adoption, a chain of foster relationships results. Long chains are undesirable from a theoretical perspective, because long chains destroy the guarantee for O(log(N)) nodes along a root-to-leaf path. From a practical perspective, long chains can be avoided by resolving foster relationships as soon as possible by opportunity or by force. An opportunity arises if a root-to-leaf traversal encounters a foster parent and the thread is able to acquire exclusive latches on both the foster parent and its parent node without delay, i.e., without waiting. If the appropriate latches are not immediately available, a thread might wait for them and thus force adoption of the foster child by the permanent parent.

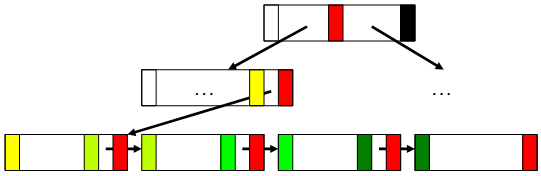

Figure 10. A chain of foster relationships.

Note that a read-only user transaction may invoke one or more adoptions, because each structural change in the index can be a separate transaction (i.e., a system transactions – see below). Thus, any B-tree traversal may perform adoption if the need and the opportunity arise. On the other hand, forcing adoption may be limited to insertion transactions, because only those incur the danger of forming or even extending a chain.

2.2 Load balancing

Load balancing among sibling nodes requires movement of records (leaf records or branch entries, as appropriate) and adjustment of the branch key in the parent node. Thus, in prior B-tree designs including B[link]-trees, load balancing requires exclusive latches on three nodes.

In Foster B-trees, load balancing is divided into three steps. First, the affected sibling nodes become foster parent and foster child. In other words, the branch key is removed from their parent node and moved into the foster parent. This can be accomplished with only 2 latches. Second, with only the affected sibling nodes latched, records are moved and fence keys are adjusted. Third, with latches only on the foster parent and its parent, pointer and page pointer are moved from the foster parent to the permanent parent.

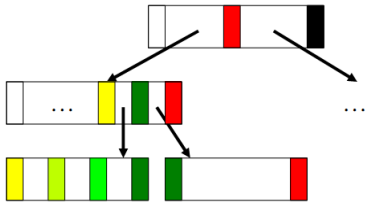

Figure 11. Initial state before load balancing.

For example, the Foster B-tree of Figure 11 might illustrate the initial state in need of load balancing among two leaf nodes. The first step does the opposite of an adoption; the result is the Foster B-tree shown in Figure 12. If the overflowing leaf node lacks free space for this transition, a split operation is required rather than load balancing. For this transition, only the permanent parent and the newly minted foster parent require latches; the newly minted foster child is not involved in this first step.

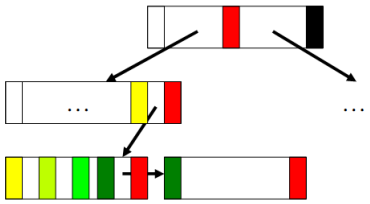

Figure 12. Preparation for load balancing.

The second step balances the load between foster parent and foster child, including adjustment of the foster key in the foster parent and of the low fence key in the foster child. The result is like the Foster B-tree shown in Figure 6. The parent node in the level above is not involved and only two latches are required, namely for foster parent and foster child. The third step brings the Foster B-tree back into a desirable steady state, i.e., it resolves the foster relationship by adoption. The result looks like the Foster B-tree shown in Figure 7. It is similar to the Foster B-tree in Figure 11, but with a different branch key in the permanent parent and, of course, different fence keys in the sibling that match each other and the branch key.

Each step can be a simple transaction including a commit record in the recovery log. In fact, the entire transaction might be described in a single log record that includes the commit. This transaction is very different from a user transaction and is therefore called a system transaction.

A system transaction does not require locks on database contents since it must not modify database contents, only the representation and data structures representing the database contents. It does not require an execution thread separate from the user transaction and therefore it does not require its own latches. (If the user transaction runs in multiple threads, e.g., in parallel query processing, each one of those acquires its own latches and invokes its own system transactions.) Finally, a system transaction does not require that log buffers be forced from memory to stable storage as part of transaction commit. (If a system transaction is not repeated during system recovery, no database contents are lost. If a subsequent user transaction relies on a system transaction, it will force the system transaction's commit record to stable storage ahead of the user transaction's log records. In that case, the system transaction will be recovered prior to the updates in the user transaction.)

2.3 Allocation details

Adding a foster child to an overflowing node can be divided into even smaller steps. The first step merely invokes free space management to find an appropriate page, formats the page to be a B-tree index node, links it to the foster parent, writes the appropriate log records, and releases all locks or latches on the

data structures used for free space management. The foster child remains empty except for two equal fence keys.

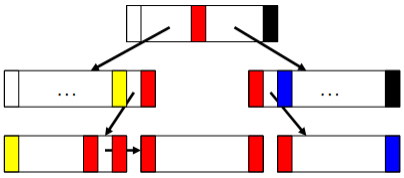

Figure 13. Preliminary state immediately after page allocation.

The result of this step is shown in Figure 13. Since one of the fence key values is exclusive by definition, two equal fence key values imply an empty key range for the node. In the diagram, both foster parent and foster child hold two copies of the foster key value; in an implementation, each may hold just one copy. This reduces space requirements in the foster parent and formatting effort in the foster child. The second step performs load balancing between foster parent and foster child. This step also determines a key value to separate the key ranges of foster parent and foster child, and it modifies the foster key in the foster parent and the low fence key in the foster child.

Each step can be a system transaction, i.e., it is logged and recovered after a system failure but it can proceed in the same execution thread, relying on the latches of the user transaction's thread, without locks, and without forcing log buffers from memory to stable storage.

Altogether, splitting a node can take as many as 6 steps:

i. A root-to-leaf search on behalf of an insertion finds a node to be full and marks it for subsequent splitting. Any subsequent operation might split the node when convenient, e.g., with respect to latch acquisition.

ii. A new node is allocated, formatted with an empty key range, and becomes a foster child of the overflowing node (Figure 13).

iii. Load balancing between the overflowing node and the new node, with appropriate adjustment of foster key and fence key.

iv. Adoption of the new node by the permanent parent, i.e., moving foster key and pointer from the foster parent to the permanent parent.

v. Reorganization (compression by additional prefix truncation exploiting the smaller key range) in the old, formerly overflowing node.

vi. Reorganization (compression by additional prefix truncation exploiting a key range smaller than in the old, formerly overflowing node) in the newly allocated node.

Each step can be an independent system transaction. Some of the steps do not require any logging. Similar to the "full" marking, there might be a "reorganization" marking that suggests reorganization (compression) when convenient, e.g., when an exclusive latch is available without delay.

Load balancing clears the "full" marking (of step i above)in the foster parent and adoption might set the "full" marking in the permanent parent if indeed the new branch key used up the remaining free space in the parent node.

2.4 Node deletion

Due to the invariant that, at all times, there is only a single (incoming) pointer per node, node deletion (and thus page reclamation) is much easier in Foster B-trees than in B^{link}-trees.

The steps in node deletion are precisely the inverse of those in node insertion, i.e., node splits. In the first step, two sibling nodes become foster parent and foster child. In the second step, load balancing moves all contents from the foster child to the foster parent. If this is not possible, then node deletion is not possible. After this step, the foster child is empty except for two equal fence keys. In the third step, the empty node is removed from the B-tree and registered as appropriate for free space management. Deletion of a leaf node might trigger underflow in a branch node, eventually followed by load balancing among branch nodes or deletion of a branch node.

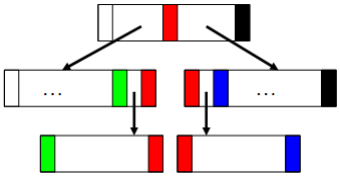

Figure 14. Starting point for node deletion.

Deletion of a branch node employs the same steps as deletion of a leaf node. Assume that one of the branch nodes in Figure 14 is in an underflow state due to a recent deletion of a leaf node. The inverse of adoption prepares the B-tree for structural changes (Figure 15), load balancing reduces the foster child to an empty key range similar to the preliminary state during node insertion (Figure 16), and deallocation transfers ownership of the page from the B-tree to free space management (Figure 17).

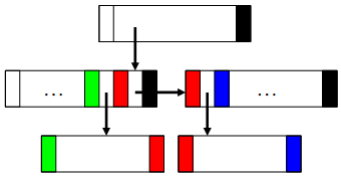

Figure 15. Before load balancing among branch nodes.

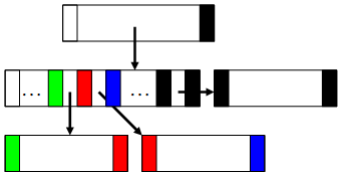

Figure 16. After load balancing in preparation for node deletion.

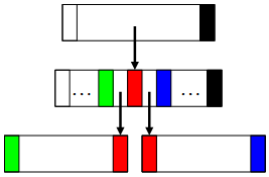

Figure 17. After node deletion.

2.5 Changes in B-tree height

If, after deletion of one of its immediate children, the root node has only a single child, and if that child is not a foster parent, then the root node can be removed from the B-tree, such that its former child becomes the root page. Thus, shrinking a Foster B-tree by a level requires very little logic, very little latching, and very little logging. Figure 17 shows such a case; the result of root deletion, a B-tree with reduced height, is shown in Figure 18.

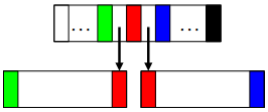

Figure 18. After deletion of the root node.

Inversely, after the root node becomes a foster parent, a new root node is needed, i.e., the B-tree needs to grow by one level. The requirement for a new root node is not immediate, because the Foster B-tree functions correctly with a root node being a foster parent.

Tree growth is divided into multiple small transactional steps with very little logic, very little latching, and very little logging, just like all other structural operations in Foster B-trees, including leaf insertion and

deletion, branch node insertion and deletion, load balancing and tree shrinking. The sequence of steps is not, however, the precise opposite of those during root deletion.

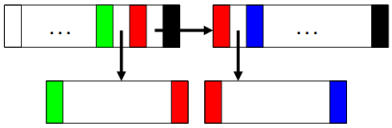

Figure 19. Root node with foster child.

The first step allocates a new page and links it to the root node as a foster child with an empty key range. Assuming the Foster B-tree in Figure 18 as a starting point, an overflow and split of the root node to the state shown in Figure 19. (The preliminary state with an empty key range in the foster child is not shown.)

The second step allocates a new root with only a single child. The result is equal to the tree shown earlier in Figure 15. The last step lets the new root node adopt the foster child. The result of this step is equal to the tree in Figure 14.

3 Summary and conclusions

Foster B-trees are defined by three invariants or characteristics. The first invariant, a single pointer per node at all times, simplifies the logic in many structural updates (compared to B^{link}-trees), in particular for load balancing among sibling nodes and for node deletion. The second invariant, two fence keys in each node, enables comprehensive yet efficient consistency checking, both offline (batch) and online (continuous). The third invariant, temporary foster relationships, enables structural modifications in small, local, transacted steps: three steps for most structure change, four steps if the B-tree height changes.

Thus, the invariants can be memorized as

i. one pointer per node,
ii. two fence keys per node (and two latches per system transaction),
iii. three system transactions per structure change, and
iv. four system transactions to change the B-tree height.

The primary characteristic of Foster B-trees, a single (incoming) pointer for each node at all times, focuses on parent pointers and implies the absence of sibling pointers. During structural modifications, a left sibling node may act as temporary parent, known as foster parent, to its immediate right neighbor, known as foster child. However, foster relationships are always temporary. They help not only with node insertion but also with load balancing and with node deletion.

Usually, each node in a Foster B-tree has two fence keys for consistency checking. While a node serves as a foster parent, it has an additional key called the foster key. The foster key is both the upper bound in the foster parent and the lower bound in the foster child, i.e., bounds on key values that may be inserted into these nodes.

In conclusion, small atomic steps enable high concurrency among execution threads on modern many-core hardware. A single pointer per node enables efficient node movement, e.g., in write-optimized ("log-structured") regimens required for efficient use of RAID and flash storage. On modern storage hardware with endurance concerns, the fence keys enable frequent and comprehensive consistency checks and thus immediate corrective action, e.g., single page recovery [Graefe and Kuno 2012]. Finally, frequent consistency checks are highly desirable for rapid development of reliable transactional indexing software.

References

[Bayer and Schkolnick 1977] Rudolf Bayer, Mario Schkolnick: Concurrency of operations on B-trees. Acta Inf. 9: 1-21 (1977).

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

[Chen et al. 1994] Peter M. Chen, Edward L. Lee, Garth A. Gibson, Randy H. Katz, David A. Patterson: RAID: high-performance, reliable secondary storage. ACM Comput. Surv. 26(2): 145-185 (1994).

[Graefe 2004] Goetz Graefe: Write-optimized B-trees. VLDB 2004: 672-683.

[Graefe and Kuno 2012] Goetz Graefe, Harumi Kuno: Single-page failures. PVLDB 5(7): 646-655 (2012).

[Graefe and Stonecipher 2009] Goetz Graefe, R. Stonecipher: Efficient verification of B-tree integrity. BTW 2009: 27-46.

[Gray et al. 1976] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, Irving L. Traiger: Granularity of locks and degrees of consistency in a shared data base. IFIP Working Conference on Modelling in Data Base Management Systems 1976: 365-394.

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

[Lomet 2001] David B. Lomet: The evolution of effective B-tree page organization and techniques: a personal account. ACM SIGMOD Record 30(3): 64-69 (2001).

[Lomet 2004] David B. Lomet: Simple, robust and highly concurrent B-trees with node deletion. ICDE 2004: 18-27.

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

[Mohan and Levine 1992] C. Mohan, Frank E. Levine: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD 1992: 371-380.

[Rosenblum and Ousterhout 1992] Mendel Rosenblum, John K. Ousterhout: The design and implementation of a log-structured file system. ACM TOCS 10(1): 26-52 (1992).