University of Kaiserslautern
Department of Computer Science
Database and Information Systems

Seminar
Optimizing Data Management on New Hardware

Summer Semester 2014

Table of Contents

Foster B-Trees

Lucas Lersch

University of Kaiserslautern
Germany

Abstract. In order to achieve performance improvement in modern system architectures, data structures for many-core processors and memory hierarchies with flash storage should be considered. Foster B-trees combine different B-tree approaches to achieve a more complete solution. From B^{link}-trees is is possible to optimize concurrent execution by dividing structural changes in small steps. Write-optimized B-trees enable large write operations on flash devices, as well as wear leveling and efficient defragmentation. Symmetric fence keys can be used for continuous and efficient verification of B-tree invariants at a lower cost and complexity. The goal of designing a Foster B-trees is to benefit from advantages from each technique in a coherent way.

1 Introduction

Even though CPU manufacturers deliver microprocessors with an increasing number of transistors, it is possible to notice that clock rates stabilized somewhere between 3 GHz and 4 GHz. The current approach to improve performance is to focus on a higher parallelism, i.e, many-core processors and multiple threading units. Since many-core processors allow an higher number of operations to be executed in parallel, it is expected that a higher processor utilization also requires a higher demand for I/O.

Despite the performance improvement of processors, the performance of traditional hard disk drives (HDD) has barely increased at all. The mechanical nature of hard disk drives implies, for each data access, waits for the disk head to seek and platter to rotate which insert millisecond pauses between microsecond data transfers. Even if we consider that the storage capacity of hard disk drives doubles every 18 months, the I/O rate of a single hard disk drive is roughly constant, meaning their performance per GB is declining. In this scenario, it is commonly the I/O performance that limits the performance of the whole system, not CPU or memory. An alternative is to add more and more hard disk drive for parallelization, however this only address the challenge if less and less data is stored per disk, making them less efficient in power and space.

Solid-state disks (SSD) arise as an alternative to HDDs and the I/O crisis in storage. Unlike mechanical hard disk drives, SSDs have no moving components, eliminating the mechanical delay of moving a disk head or rotating a platter and consequently eliminating the primary roadblock to high performance and energy efficiency. By offering a much better I/O performance than HDDs, SSDs accelerates data delivery to the processing cores of many-core processor, which can dramatically reduce latency and improve utilization for data-intensive applications. It is possible to note the obvious synergy between many-core processors and new memory hierarchies based on SSDs.

Such synergy changes the traditional systems bottlenecks and creates the need to rethink the way common data structures are implemented in order to better explore the advantages offered by these new technologies. In this context, the B-tree data structure plays an important role, since it is widely used for storing sorted data and allowing searches, insertions and deletions at logarithmic time. File systems and

database systems have being employing B-trees for implementing access paths and indexes for so many decades that the data structure can be considered ubiquitous in these environments. There is a need to reconsider the B-tree data structure aiming at minimizing the concurrency requirements in order to make them more suitable for many-core processors. It is also important to exploit SSDs inherent storage characteristics for achieving the best performance when storing a B-tree data structure on such devices.

This work examines approaches to achieve a better performance of B-trees considering a many-core processor environment with SSD storage and finally describe Foster B-trees, a variant of B-tree that aims at combining these approaches. Section 2 will review basic B-tree concepts and terminologies. Sections 3, 4 and 5 are going to motivate and discuss three different B-tree variants. In Section 6 we are going to describe how these variants can be combined in order to create a Foster B-tree.

2 Background

2.1 Latches and Locks

In order to provide a better understanding of B-trees and to use the appropriate terminology it is necessary to clarify and distinguish two different concepts present in the database literature, but often referred by the same name: locks. [GR92] approach these two concepts when discussing synchronization in B-tree indexes from a page-oriented perspective and from a tuple-oriented perspective. The difference here becomes clearer if we consider these two views as the database physical representation (in disk pages) and the database logical contents representation (in tuples), respectively. At this point we consider that latches are used for synchronizing the access to shared data structure in memory (the image of a disk-based B-tree node, for example) by multiple threads being executed in parallel. Locks, on the other hand, are used for coordinating the access to database contents and its logical representation in B-tree indexes by multiple concurrent database transactions. Latches and locks are represented in different implementation primitives in order to achieve the desired behavior.

Latches are usually implemented by a semaphore which usually offers shared and exclusive modes. A latch is usually embedded in the data structure meant to be protected by it. They are designed to be fairly simple and to offer minimal overhead for being acquired, maximal performance and scalability. Deadlock avoidance is desired and achieved by appropriate coding disciplines. To this end, once a thread acquires a latch, it is meant to hold it for a very short duration, only during the required time of a critical section. In order to keep a small critical section granularity, latches should only be requested after all the data required by the thread has been already loaded in the memory, avoiding a thread to hold a latch during disk I/O. In a database system, latches usually protect data structures such as a buffer management table, a lock manager table and B-tree nodes (represented by disk pages). In the case of a B-tree indexes, latches protect operations that modify its physical structure (node splitting, load balancing among nodes, creation and removal of ghost records) but not its logical contents.

Different from latches, locks are generally complex to implement. By using a hash table, the lock manager is able to manage the locks needed for coordinating parallel transactions querying and modifying database contents. Once a transaction acquires a lock, it should hold it until its end. It is important to note that the lock hash table is an in-memory data structure, and therefore, the access to it is protected by latches. Lock modes can range from simple read and write modes to a wide variety of combinations with additional intention modes. By employing these modes the lock manager is able to apply sophisticated scheduling policies. This level of sophistication, however, implies a very expensive lock acquisition operation. In contrast to deadlock avoidance, locks usually enable the detection of deadlocks and resolution by rolling back prior actions. In the processing of a query of a transaction, locks are needed not only to guarantee the presence of data, but also to guarantee the absence of data. In this context, techniques as key-value locking and key-range locking are employed to achieve such guarantee. In key-range locking, a transaction that locks a key implicitly locks the whole interval from this key to the nearest higher key (with open interval at the nearest higher key) in order to guarantee that no key in this range is going to be modified. Table 1 shows the main differences between locks and latches.

In the remaining of this work we are going to use this terminology of latches and locks to refer to these two different concepts. Since most of the topics to be discussed in the following sections are related to physical representation and organization of B-tree indexes, the concept of latches will be explored more frequently than locks.

	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

Table 1. Locks and latches.

2.2 B-Trees

In this section we are going to define the basic B-tree structure and operations on which the subsequent sections are going to be based to introduce new variants of B-trees.

Motivation B-trees differ from other tree structure by its characteristic that each node has many successors. Moreover, because fairly easy to keep a B-tree balanced in terms of nodes, its main property is that each search path has the same length when considering the number of nodes. The guarantee of small average time for retrieval, insertion and deletion of entries make a B-tree a quite appealing data structure. For this reason, during decades, B-trees are the chosen data structure for implementing indexes in database systems that map a search key to associated information [GR92].

Data Structure Each node of the B-tree has the same fixed size and is physically represented by a disk page. There are two types of nodes: internal nodes and leaf nodes. Internal nodes act merely as redundant information that enables efficient search operations. All logical data contents (tuples) are stored in the leaf nodes. A B-tree node contains a set of entries and each entry is represented by a record inside a disk page. An entry in an internal node is a pair in the form <K, P>, where K is a dividing key and P is a pointer to the sub-tree containing information about all keys values larger than K (called a parent-to-child pointer). An entry in a leaf node is similar, but instead of a parent-to-child pointer, it contains the data which is organized in the B-tree. In additional to the incoming parent-to-child pointer, leaf nodes also have two other incoming pointers from the lower and higher neighboring leaf nodes, in such a way that all the leaf nodes in the B-tree form a doubly linked list. Entries in a B-tree node are sorted in ascending order by the key-value in which the index structure is based on. For better space utilization, entries can have a variable length, considering that the length of dividing keys in a B-tree index is variable. Figure 1 shows the basic structure of a B-tree index where shadowed areas represent dividing key values.

Additional concepts either do not impact the discussions that take place in the following sections or are going to be introduced in the following sections in order to provide a better understanding of the B-tree variants to be introduced. Among the concepts to be ignored here are: detection and treatment of underflow of a node, removal of empty nodes, node compression (suffix truncation and prefix truncation), type of information associated with B-tree keys.

Since a B-tree is desired for implementing a database indexes, it is necessary to guarantee that the operations on the B-tree data structure will respect the ACID properties required for tuple operations. In order to achieve context isolation of

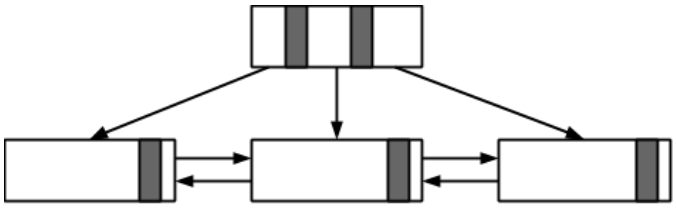

Fig. 1. Basic B-tree structure.

multiple threads running in parallel, all operations on the B-tree data structure must be protected by latches. In order to achieve optimal performance, the number of acquired latches and their holding time must be reduced as much as possible. For example, a general rule is to release a latch as soon as the routing information of a internal node has been exploited. These characteristics must be taken in account when defining the operations that act on a B-tree.

Retrieval "Latch coupling" is the most known technique applied for retrieving an entry from a B-tree, given a key. It consists of latching the root node in shared mode, searching for the pointer to follow, acquiring a shared latch on the child node indicated by the pointer and then releasing the shared latch of the father node. This crabbing through B-tree nodes proceeds during the whole root-to-leaf traversal until the desired leaf node is reached. In the case of a range-query in a non-unique index, multiple entries (tuples) must be retrieved from the leaf nodes. To do so, the crabbing technique is also used to move from a leaf node to the adjacent leaf node using the pointers that form the linked list at the leaf level.

A thread needs to acquire a shared latch while reading a node searching for the appropriate pointer to follow in order to guarantee that no other thread is going to modify the page contents during this search. Requesting a shared latch on the child node prior to releasing the shared latch of the parent is required to guarantee that the parent-to-child pointer is not going to be invalidated by another thread that can deallocate the child node or balance the load among neighboring node, for example. The case where two threads perform ascending and descending traversal on leaf nodes should be considered and carefully treated in order to avoid deadlocks. If necessary, one of the threads should release its latches to allow the other conflicting thread to proceed and then retry the whole root-to-leaf search.

Finally, after we have understood the basic behavior of a retrieval operation, we have to consider the case where the child node to where we have to navigate during a search is not in the buffer pool. In this case, I/O is required for the node to be loaded from disk into the buffer pool in order to be modified. However, as mentioned in the previous Section 2.1, latches should not be held during I/O and then the latch on the parent node should be released prior to the I/O. If possible, before releasing the latch on the parent and before requesting I/O operation, a latch can be acquired on the descriptor of the node page to be loaded from disk in the buffer pool. A second approach is to release the latch on the parent node, request I/O operation and restart the root-to-leaf traversal. In this last case, instead of restart a whole new root-to-leaf traversal, it is possible to optimize by resuming the original one verifying the log sequence number of the node pages of the traversal to guarantee that they were not modified while the requested node page was fetched from the disk.

Insertion For inserting a new entry in a B-tree a special case must be considered: the node where the entry has to be inserted is full and must be split in order

to accommodate the new entry. A node split in a B-tree can be performed by different approaches. The most naive and less efficient one is to acquire a exclusive latch on the entire B-tree structure. To provide better performance in a concurrent environment, a more practical approach is required.

The common approach that is considered in this work is optimistic [BS88], i.e. it assumes that a node splitting will not happen as a side effect of the current insert operation. The initial steps of the insertion proceed exactly the same way described for the retrieval operation. A shared latch is acquired for the root node and by crabbing through the B-tree nodes it is possible to determine the correct leaf node where the insertion of the entry will take place. At the leaf level a exclusive latch should be acquired, since we are going to perform an update by adding a new entry. Additionally, the sequence of nodes of the root-to-leaf traversal is stored in a auxiliary structure that will be referred later. If the leaf node where the new entry has to be inserted is not full, then no split is necessary. In this case, the insertion of the new entry in the leaf node takes place and the exclusive latch of this leaf node is released.

However, if the leaf node where the new entry has to be inserted is full, a split operation is needed. Another optimistic assumption is made to provide : the path from the first root-to-leaf traversal did not change. The exclusive latch acquired for the leaf node is released to avoid deadlocks and the path that was stored in the auxiliary structure during the first traversal is retraced, this time acquiring a exclusive latch on each node. Crabbing proceeds by acquiring exclusive latches on the successor nodes, and releasing the exclusive latch on the current node if the successor node is not full, because then there is a guarantee that the successor node can accommodate new entries without the need of splitting and the split will not propagate up to this level. As a result, all nodes from the leaf level upward that are completely full, plus one level above, will be latched on exclusive mode. At this point the split will take place: a empty page is allocated, the upper half of the entries of the full node are moved to the new node, forward and backward pointers between the original node and the new node are adjusted, a new separating key (the lowest key in the new node) and parent-to-child pointer is inserted in the parent node (which is still latched on exclusive mode) and finally the new entry is inserted into one of the two nodes resulting from the split. Since, after the split, we are going to insert a new separating key and parent-to-child pointer in the parent node of the full node, the parent node might need a split also. The split will then propagate its way up to the B-tree until it finds a node which can accommodate a new separating key and parent-to-child pointer. In the worst case scenario, a split will propagate all the way up to the root node and require the root node to be split. The approach in this case is similar and will provoke a growing of one level in the B-tree height. It is important to note that a node split on higher levels of the B-tree only occur as a consequence of a split on a leaf node.

We can check if our second optimistic assumption was wrong and that the path of the first traversal has changed by comparing the pages log sequence number while going down the second time on the B-tree. In this case a new root-to-leaf traversal must be executed in order to find again the leaf node where the entry has to be inserted and retry the operation.

Deletion The procedure to delete an entry from a B-tree is similar to the insert operation. It starts with a root-to-leaf traversal to determine the node where the entry to be deleted is located, acquiring shared latches during crabbing through internal nodes and a exclusive latch at the leaf node. The entry is deleted from the leaf node, the remaining entries of the leaf node are re-organized and the exclusive latch of the leaf node is released. Even if the key of the removed entry served as a

separating key in a internal node, the key does not need to be removed from this internal node. Keys in internal nodes only specify a range of key-values to correctly guide the search operation down to the key values in the leaves.

Analogously to a node split operation that happens as a consequence of an insert operation, delete operations may drop the occupancy of a node below a certain threshold (50% of the maximal node capacity, for example) and create the desire for merging the entries in this underflowing node with other nodes. Such desire for merging nodes are motivated not only by concerns about storage costs, but also by performance of B-tree operations. Removing B-tree nodes as a consequence of merge operations may reduce the number of internal nodes required for creating the routing path for a given number of leaf nodes. The fewer the number of internal nodes, the lower the B-tree height can be and consequently the faster is a root-to-leaf traversal.

However, merging nodes has shown to be a very complex and expensive operation, to a point that synchronization requirements for restructuring the B-tree can significantly reduce the parallelism. These characteristics also apply when considering a split operation, but there is nothing that can be done to avoid a split if we want to keep the B-tree correct and operational. Unlike a split, in the case of an a underflowing node we have a very efficient alternative to merging the node: not merging it. The motivation for avoiding merge operations comes from the fact the a database data volume tends to increase rather than decrease, and so future insert operations will guarantee that any node remains in a underflowing state only during a short period of time.

In the case of not merging underflowing nodes we can allow a node to be completely emptied by delete operations. The retrieval and insert operations must be able to handle empty nodes in this case. In order to promote space reclamation, a asynchronous clean-up utility may run during low system load time to remove completely empty nodes from the B-tree. While removing a completely empty node from the B-tree, this utility must remove the entry with the respective key from the parent node (removing the parent-to-child pointer as well). Deleting this entry from the parent node may provoke the parent node to be empty and thus provoke it to be also removed from the B-tree, similar to a split operation that propagates way up to the tree. However, if a propagation of node deletions happen, it is possible to divide it into multiple asynchronous steps. Additionally, if the node to be deleted is a leaf node, the forward and backward pointer of the neighbor nodes must be adjusted accordingly. By following this approach, the delete operation itself becomes extremely cheap.

3 B^{link}-Trees

In this section we produce the concepts introduced by [LY81] to improve parallelism of concurrent B-tree operations. Considering the base B-tree approach defined in the previous section, we are going to point the differences of B^{link}-trees.

3.1 Motivation

In addition to provide a B-tree structure that is concurrently and correctly manipulated by several threads, it is highly desirable to make this manipulation with operations as optimized as possible. Modern many-core processors can achieve higher concurrency of threads if we are able reduce the latching requirements by reducing the granularity of critical sections and atomic operations. The B-tree defined in the previous section suffers from a certain level of latching complexity and unpredictable latch retention times. In order to better exploit the multiple cores of a processor, B^{link}-trees proposes operations with a reduced number of latches allow these operations to be split in smaller steps.

3.2 Data Structure

The data structure of a B^{link}-tree is very similar to the base B-tree defined in the previous section. Very small modifications are made in the data structure in order to enable the optimized operations that are going to be defined in the following sections. For a better understanding, first we consider that the forward and backward pointers between neighboring leaf nodes are eliminated, so the nodes at the leaf level are not chained anymore. To this point, each node has only one incoming pointer, the parent-to-child pointer. Second, we introduce an additional link pointer to each node. The link pointer of a node points to the next node at the same level of the tree. It is important to note that, unlike the previous approach, the link pointer flows in only one direction (forward or backward, depending of the implementation) and there is a chain of nodes at each level of the B^{link}-tree, not only at the leaf level. The purpose of this link pointer is to provide an additional method to reach a B^{link}-tree node, rather than only the parent-to-child pointer. Each B^{link}-tree node has now two incoming pointers, the parent-to-child pointer and a link pointer from one of the immediate neighboring nodes at the same level (depending of the direction the link pointer flows).

3.3 Retrieval

The retrieval operation of a B^{link}-tree proceeds in a very similar way as the one previously defined, but with two main differences.

First, it should consider the link pointer in each node. The retrieval starts by searching the root node for determining which pointer it should follow to a child node in order to find the entry with the desired key value. The crabbing through a root-to-leaf traversal proceeds as expected. However, if at a certain node the highest key value is lower than the one we are searching, it means that this node was split and this split was not reflected in the father node by the time the father node was searched. The operation is then at a wrong node and instead of restarting the whole root-to-leaf traversal, it can follow the link pointer of the current node in order to find the correct node to resume its search.

A second difference mentioned in [LY81] is the advantage that no shared latches or exclusive lathes are acquired and latch coupling is not required during the retrieval of a node. This entails the fact that no search is ever prevented from reading

any node. However, the storage model considered in [LY81] assumes that each thread reads and writes disk pages from and to the disk directly using atomic operations (get and put, respectively). In this scenario, each execution thread has its own copy of a disk page. Although, since most modern systems employ buffer management, multiple threads will access the same page structure of a node and this page structure resides in the buffer pool. If this is the case, shared latches and latch coupling are required in order to read a node, just like explained in Section 2.2 and this second difference does not exist.

3.4 Insertion

The main advantage of B^{link}-trees is that it allows higher parallelism by dividing an insert operation in two independent steps. This optimization is made possible by using the link pointer of nodes.

The first step is similar to the classical insertion described previously. We acquire a shared latch on the root node and perform a root-to-leaf traversal with latch coupling in order to determine the node where the new entry must be inserted. This traversal should consider the link pointers, as mentioned in the retrieval operation in Section 2.2. The path from the traversal is also stored in a auxiliary structure. When the leaf node is found, a exclusive latch is acquired and if it supports (has enough space) the new entry, the insertion is made, the exclusive latch is released and the operation ends. However, if the leaf node is overflowing, splitting it is required in order to create space for the insertion. Instead of releasing the exclusive latch on the leaf node and retracing the root-to-leaf path acquiring exclusive latches, we allocate a new node (disk page) and move the higher half of the original node to the new node. The link pointer in the new node will point to the same node pointed by the link pointer of the original node. The link pointer of the original node will point to the new node. The new entry is inserted in the appropriate node (the original one or the new one) and the exclusive latch on the original node is released. The first step ends here. It is important to note that a new entry was not added to the parent node as a consequence of the creation of a new node. However, the new node is still accessible by the link pointer of the original node. A important requirement is that the original node must occupy the same disk page it occupied before the split took place, so no modification in the parent node is required. Here, the original node and the new node act as a single node and this intermediate state is supported by the B^{link}-tree structure in order to provide correct operations.

This arrangement after a split operation should be only a temporary fix to allow correct concurrent operations. A chain of nodes linked only by a link pointer may decrease the performance of search operations in the B^{link}-tree, and even if this state happens only as a consequence of a node splitting, which tend to be infrequent operations, it is necessary to insert a new entry on the parent node as early as possible in order to guarantee a optimal B^{link}-tree structure.

The second step consists of restoring the B^{link}-tree from its intermediate state to a proper state and can be executed completely independent of the first step. Using the stored path of the first root-to-leaf traversal we retrace the way back up to the tree, latching on exclusive mode the parent node and insert a new entry (key and parent-to-child pointer) in the parent node. If the parent node also requires a split, we repeat the operation up to the tree until we find a node that can accommodate a new entry. In the end we release all the acquired latches.

The main gain of the insert operation in a B^{link}-tree is that a small and constant amount of latches need to be acquired by any thread at any time. If we analyze the latching efficiency of the insert operation, in the best case, where no splitting of a node is required, only the leaf node is latched on exclusive mode. The worst case scenario happens when the leaf node requires a split, we move up to the parent node

and latch it on the exclusive mode, but another thread has already modified the parent node and it was split. If the key of the entry we have to insert in the parent node is higher than the highest key of the parent node, then we need to follow the link pointer of the parent node to a new node, also acquiring a exclusive latch on it. In this case, a single insert operation latches three nodes: leaf node, parent node and a node at the same level of the parent node while we follow a link pointer. The worst case scenario tends to occur rarely in a B^{link}-tree with large capacity on each node.

3.5 Deletion

The delete operation is proceeds the same way as the one described in Section 2.2, allowing a node to be completely emptied. The only change is made in order to support search of node considering the link pointer relation, as described for the retrieval operation for B^{link}-trees, in Section 3.3.

3.6 Livelock

The operation in B^{link}-trees described previously do not avoid the occurrence of livelock where a thread runs indefinitely because it keeps following link pointers to nodes created by other threads. However, in modern systems, such scenario is very unlikely to happen. If the threads that request a latch are coordinated by a scheduling policy that assign priorities considering the age of the threads, this situation can be avoided.

4 Write-Optimized B-Trees

4.1 Motivation

In database systems, the dominant I/O patterns are reads of individual pages base on index lookups and writes of updated versions of those pages. If in the past, we had an absolute majority of read operations, in modern system architecture we can notice that memory size grows, consequently the fraction of write operations tends to increase. This happens because a larger memory implies an increased hit ratio in the buffer pool and less read operations are required. For this reason it becomes important to increase the performance of write operations.

4.2 Flash Memory

Modern flash-based devices such as solid-state disks (SSD) enabled important improvements in database systems and file systems. The main advantage of SSDs is that, due to the lack of mechanical parts in contrast to hard disk drive, it allows persistent data storage with lower access time and less latency. Modern systems often make use of SSDs in order to compose a memory hierarchy that improves the response time of the system as a whole. However SSDs are NAND-based flash memory, which adds new requirements that must be considered when utilizing data structure on these devices.

Flash memory suffer from two main limitations. The first one is that, even if it allows random-access read operations and writes to be performed in a single page, erase operations can only be performed in block units. For this reason, in order to update a single data page it would be necessary to erase the whole block. This contrasts with additional HDDs, where it is possible to write and erase data directly in any particular sector.

The second limitation is that flash memory has a finite number of erase cycles, meaning blocks wear out after a certain number of erases. Therefore, it is not desirable to erase a whole block of data to overwrite only a single page (as mentioned by the first limitation). The main approach to address this problem is to promote wear leveling by distributing writes as evenly as possible across all the blocks in the flash memory. In a perfect scenario, this would enable every block to be written to its maximum life, so they all fail at the same time. To achieve this, every time a page is updated, it is rewritten to a new location in a previously erased block and the data in the old location is invalidated. This differs from HDDs, where it is possible to overwrite updated data in the same disk sectors the old data is.

Write-optimized B-trees [Gra04] attempt to increase write performance by enabling large write operations that can be as large as entire erase blocks of a SSD. Furthermore, wear leveling can be guaranteed at the same time due to inexpensive page migration operation.

4.3 Log-Structured File Systems

The idea behind log-structured file systems is to replace multiple small write operations by a single large write operation in order to increase write performance. The two main advantages of this approach are the reduced number of seek operations and the reduced overhead related to maintenance of parity information in disk arrays with redundancy. These are the characteristics that the design of a write-optimized B-tree aims to replicate.

However, in order to enable a single large write operation, log-structured file systems requires the flexibility to write dirty pages to an entire new location. This entails two new costs. First, there is the need of a mapping layer that maps a page

identifier to its physical location on disk. This mapping layer introduces overhead, because a mapping structure requires additional I/O, locking, latching and searching. Second, since updated pages are written to a new location, old pages become obsolete while still occupying additional disk space. This second cost is justified by the fact that modern disk technologies tend to increase storage capacity much faster than I/O bandwidth, so improving bandwidth in detriment of storage space is a good trade-off. Furthermore, assuming that most systems operate with a disk in a less-than-full state, there is an opportunity to apply efficient space reclamation of free space.

An additional aspect should be considered if we want to implement a B-tree index based on a log structure file system: transactional processing. Unfortunately, log structured file systems have the limitation of not implementing proper atomicity and durability in order to guarantee the ACID properties. As an example, suppose two concurrent transactions updates tuples of the same page in a database system that supports locking granularity smaller than a page. If one of the transactions commits and the other one rolls back, there is no image of the page in a correct state. For this reason and the two inherent additional costs, log-structured file systems were never before successfully used on database systems.

To overcome the imposed limitations, the approach is not to enable log-structured file systems to support B-tree indexes, but to integrate the large write operation of log-structured file systems into already known B-tree structures. Additionally, the overhead related to locating a page using a mapping layer is already similar to that in traditional B-tree indexes. Furthermore, since updated pages are written to new locations in order to improve write performance with large writes, high levels of fragmentation on the disk and decrease the performance of sequential scans (which are very important in database systems) may occur. As a consequence, there is no overall system performance gain. Since scan operations are very frequent in database systems, there is a need to enable large writes without impacting the performance of scans. The design of write-optimized b-trees address this problem by allowing that certain indexes and tables to be updated in-place (read-optimized). A policy can be used in order to determine whether the write of a page is going to be a part of a large write operation or an in-place update.

4.4 Page Migration

As mentioned in the previous section, whenever a node in a write-optimized B-tree is updated, we can write the node back to the disk by updating the node page in the same disk location or by performing a large write operation that contains this node page, and in this case the node page is migrated to a new disk location.

Node split operations and node updates in general combined with large write operations may cause a certain level of fragmentation on the disk. However, it is highly desirable for an index structure to perform efficiently in order scans for large range queries. For this reason, it is highly recommended to perform index defragmentation on the disk periodically. A index defragmentation consists of moving B-tree nodes in a way to keep them ordered and in close proximity to the preceding node. Again, the basic operation to be performed by a index defragmentation is page migration.

Furthermore, since large write operations move a updated node to a new location, the previous location contains old data and should be invalidated. However, even if the old data is invalid, it still occupies space on the disk. In this case it is necessary to reclaim and consolidate free space. Here the problem is that node pages are freed individually, but a large write operation requires a large array of free pages. To address this problem, we can keep track of array of pages with few remaining valid pages and artificially update these valid pages without modifying their contents. As a consequence of this artificial update, this few remaining pages

are going to migrate to a new location on the disk as part of a large write operation. The old version of these pages can be freed and the whole array of pages can now lend itself for another large write operation.

It should be clear here that the most frequent operation in a write-optimized B-tree is the page migration, since it happen as a consequence of page updates, index defragmentation and space reclamation and consolidation. Hence, it is wise to have an efficient page migration strategy. A system transaction can be used to implement a page migration procedure, since system transactions have the characteristic of not modifying database contents, only its representation on disk.

Finally, since the root node of B-tree indexes are usually record in databases catalogs, the special case of a migration of the root node should be considered. One out of two alternatives may be employed. The first is to have a special system transaction to migrate the root page and the database catalogs are updated as part of the transaction. The second approach which can be employed in the case where the root node was updated is simply not to migrate the root page and write it back to disk in the same location (update in-place). Both cases require the root page to be marked in order to be properly identified by the buffer manager.

In the next sections we discuss which data structures and algorithms are employed in order to make the system transaction for a page migration as efficient as possible.

4.5 Symmetric Fence Keys

If we consider support for large write operations in the B-tree defined in Section 2.2, whenever a leaf node is updated, the write of this node page to disk may take part in a large write operation. In this case a page migration is necessary and updates are also required in the three surrounding nodes (parent, backward neighbour and forward neighbour) in order to guarantee that the incoming pointer from these nodes will refer to the new disk location of the updated node after the page migration. Since the parent node and the two neighbours nodes were also updated, when these nodes are written back to disk as part of a large write operation, each node page will be written to a new disk location and requires updates of their respectives parent node and two neighbours nodes. This situation will cause updates and large write operations to propagate in all directions of a B-tree index.

To address this problem the backward and forward pointer on each leaf node are replaced by symmetric fence keys. Each B-tree node stores an entry for a lower fence key and an additional entry for an upper fence key. The fence keys of a node define a range of key values that can be inserted in this node. The range of key-values has an inclusive bound and an exclusive bound, with the difference that the entry for the inclusive fence key can be a normal entry, but the entry of the fence key for the exclusive bound is an invalid entry (cannot be used to store a tuple). In the initial root node, the value of the lower fence key is negative infinity and the value of the upper fence key is positive infinity. Whenever a node is split, the new key value that is inserted in the parent node as a consequence of the split is going to serve as lower fence key in the new node and as upper fence key in the previously overflowing node. Replacing the backward and forward pointers by symmetric fence keys may imply some concerns.

Since the pointers among leaf nodes allow efficient range scans in the index, eliminating them might seem like a bad decision. However, in modern high-performance systems, this chain of linked leaf nodes becomes useless. Assuming that internal nodes compose only a very small fraction (around 1%) of the total amount of nodes in a B-tree indexes, systems with larger memory tend to retain all or most of the internal nodes pages in the buffer pool at nearly all times. Furthermore, many-core processors and multi-disk architectures make use of asynchronous read-aheads to

anticipate the read of index pages from the disk into the buffer pool before they are directly requested. This allows computation and I/O operations to overlap, taking full advantage of both the CPU and the disk and also collaborating for reduced frequency of buffer faults. Since information about the pages to be read are required to employ read-ahead, it is not possible to rely on backward and forward pointers. Only interior nodes can be searched in order to supply the necessary information. To this end, it is expected that the prefetching of leaf nodes pages must be guided by the parent-to-child pointers from the parent nodes or even the grandparent nodes.

A second concern is that symmetric fence keys may induce additional space requirements when compared to backward and forward pointers. Pointers occupy a fixed space where key values can be lengthy string values. Suffix and prefix truncation of the key values can be employed in order to reduce this additional space costs. Further techniques can be used to compress the nodes information on the B-tree. The entry of a exclusive fence key can be used to store information about the most frequent key value or the largest duplicate key value within the node, for example. It is important to mention that symmetric fence keys are not required to solve the problem introduced above and that alternative techniques can be used in order to avoid the additional space cost due to on-disk redundancy. However, symmetric fence keys are the chosen technique to replace the backward and forward pointers because it also facilitates the verification and diagnosing of hardware and software errors, as we will see on Section 5.

An unrelated but important additional concern is related to accessing the parent node in an efficient way. Three approaches may be used. First, a search in the B-tree index is performed in order to identify the parent of a node. If the parent page is not present in the buffer it is possible to update the node page in-place on disk or simply abandon the page migration (if this is the case). A second approach introduces the requirement of having for each node in the buffer pool its parent node also present in the buffer pool (and transitively the entire path to the root). Each node page in the buffer pool has a link pointer to the parent node also in the buffer pool. However, this introduces the need for reference counting, since multiple nodes have the same parent. The split operation of a node also imply additional complexity for linking child nodes to the newly allocated parent node. The third approach is the preferred one, since it combines the efficiency of accessing the parent node without hard requirements. It consists of linking nodes in the buffer pool to their respective parents also in the buffer pool. However, if the parent is not present in the buffer pool, it is possible to write the node page to disk updating it in-place or abandon a page migration (if this is the case). An additional cost is introduced for probing the buffer pools hash tables in order to link child pages when a parent is loaded into the buffer pool as well as remove the links when a parent is removed from the buffer pool. It is worth noting that neither of these approaches requires I/O.

Finally, symmetric fence keys allow us to get rid of the backward and forward pointers and avoids the need of updating neighbour nodes. This contributes for simplifying the page migration operation. An additional storage cost must be paid, but it can be reduced by compression techniques. Removing the backward and forward pointers of leaf nodes does not impose a relevant performance impact for index scans, since most modern systems architectures count on many-core processors and multiple disks to employ read-ahead.

4.6 Logging of Page Migrations

In order to have an efficient and inexpensive system transaction for page migration, there is the need to reduce the costs of logging. The standard logging approach is to write the node page contents as part of the log information. In order to recover from a system crash, the page can be restored simply by copying the contents previously

written in the log. However, this fully logged approach induces high logging costs for large write operations, since it requires logging the whole content of a node page for each time a node page is migrated to a new location.

The second approach avoids the high costs of logging the page contents. A small log record is used to capture the whole system transaction, including transaction begin, allocation changes, page migration and transaction commit. This small log record contains the old page location and the new page location. In order to protect the page contents, it is required that the system transaction forces the write of the page to its new location prior to committing (i.e. writing the log record to stable storage). In other words, write-ahead logging is not employed for the page migration. The basic steps for migrating a page migration: allocate disk space, migrate the page to the newly allocated disk space, write log record, update global allocation information. Note that write-ahead logging does not apply for updating the global allocation information. This is required in order to guarantee that the old page location is not going to be deallocated and that we can recover the old page contents in case of a system crash before the system transaction commits. In addition to having a single and small log record, a second benefit is that writing the log record to stable storage can be done asynchronously. These two benefits make logging overhead of a page migration truly minimal, at the expense of forcing the page contents to the new location before the system transaction can commit. This forced write approach is the preferred to be implemented in write-optimized B-trees.

The final approach is similar to the forced write approach mentioned above, with the difference that the system transaction does not require the page contents to be written to its new location prior to committing. Instead, it relies that even after deallocating the old page location, the contents will not be overwritten until the page was written to its new location. A write dependency must be implemented in order to guarantee this requirement for recovery purposes. However, this method has a potential weakness in environment that employs backup and recovery. Assuming that the backup is made online and only for pages currently allocated for a certain table or index, it is required that the operations of backup and page migration interleave in a particularly unfortunate way. If we consider that a backup of the old page location is not going to be made (since it has already been deallocated), it is necessary to complement the log record with the updated contents of the page. In this case, the behavior is the same as in the fully logged approach.

5 Verification with Symmetric Fence Keys

5.1 Motivation

It is expected for a B-tree index to become inconsistent due to hardware defects, software defects and environmental problems. In addition to mechanism employed by lower system layers (operating system, softwares, drivers, network, storage), a database system must provide on top of these its own mechanism for verification and detection of failures and corruptions [GS]. Since it is highly recommended to employ verification as part of regular system maintenance, efficient algorithms must be employed to allow this frequent testing. This work focus on the verification of the physical integrity of a B-tree structure rather than in the logical verification.

For verifying the physical integrity, a careful traversal of the whole B-tree structure could be employed to verify all in-node invariants as well as to verify all cross-node invariants (key values, pointers). However, this approach is only affordable for offline verification. It is necessary an online approach to promote an efficient and non-disruptive verification. An algorithm that requires little memory, is simple and fast could be used to verify the consistency of a B-tree index as part of a root-to-leaf traversal. Verification could also be performed during backup, which would also increase the confidence and the value of the backup.

Verification of in-page contents can be done simply by using a checksum operation of all the data in the page or only a fraction of it. The main problem faced by verification algorithms is the complex web of pointers of B-tree structures, i.e., parent-to-child pointer, backward pointer, forward pointer. The key ranges and separator keys of nodes should also be considered. In order to detect corruptions that manifest themselves as inconsistent B-tree indexes, a complete verification is required and a successful completion guarantees complete consistency of the data structure.

The output of a verification algorithm can vary between two extremes. A binary output can be used to indicate whether or not a B-tree index is corrupted. A detailed output may contain specific information and identification of each individual problem. A trade-off between performance and output information applies here and should be considered in order to apply the best solution for each situation. The following sections describe different algorithms for a structural verification of a B-tree. Furthermore, the algorithms described here assume that there are no concurrent transactions updating the indexes and tables being checked, i.e. verification operation holds a table-level shared lock that covers the indexes.

5.2 Index Navigation

In order to verify the B-tree index structure, it is required to cover the network of pointers as well as the ordering of key values among sibling nodes and among parent and child nodes. The most naive approach is to navigate the whole index structure, from the lowest key to the highest key (depth-first), matching backward and forward pointers and key ranges at all B-tree levels.

The main advantage of this algorithm is its simplicity. However, the performance is linear with the number of pages allocated for the B-tree index. If an entire table fits into the available buffer pool, I/O performance is not affected. Nevertheless, such scenario is not common and consequently repeated read operations for each page may be required, deteriorating the overall performance and scalability.

5.3 Aggregation of Facts

For terabyte databases it is unlikely to have tables that fit in the buffer pool. In these cases aggregating facts from B-tree nodes are more suitable. The verification

algorithm can be divided into two phases. In the first phase each data page is read and information is extracted in the format of facts, but the data pages are not verified immediately after read. The second phase stream these facts in a matching algorithm. If a match between two facts is found, verification of these facts were successful. All extracted facts must be matched for the B-tree structure to be consistent.

Multiple facts can be extracted from B-tree nodes. The fact "node X follows node Y" is extracted from both page X and page Y and is used to verify the chain of pointers among nodes at the same level (leaf nodes usually). The fact "node X at level N+1 has child Y for key range [a,b)" is extracted from a parent node and matches the fact "node Y at level N has keys in the range [a,b)" which is extracted from a child node. For matching facts extracted from the root page, an additional fact can be created based on the system catalog to provide an artificial parent node for the root.

Introducing information about the key range of a node to the fact "node X follows node Y" could be used to verify that all keys in X are greater than keys in Y. This is not required because this verification is already transitively verified by the separator key in the parent of both X and Y. However, two neighbor nodes might not share the same parent, but share a higher ancestor. In this case they are called cousin nodes. Cousin nodes introduce the problem of verifying that the keys among them are ordered, i.e., that all keys in the successor node are larger than the keys in its cousin node. Since there is no shared parent, it is not possible make this verification transitively, even if the key relationships between neighbors and between parents and children are correct (key relationship across skipped levels are not guaranteed).

Symmetric fence keys help solve the cousin problem. For the sake of remembering, each node has a low and high fence key that are basically copies of the separator keys presented in the parent node. The fence keys take the role of backward and forward pointers, replacing the traditional page identifier with a search key. A separator key from the root is replicated along all the path to leaf level in neighboring nodes (be them siblings or cousins). The facts extracted from nodes with symmetric fence keys have a single format: "node X at level N has key value V as low/high fence key". Each fact extracted from a node should be matched to a exact copy of the same fact that will be extract from the parent node. For this reason, only equality comparisons between page identifiers, keys and level information are required in order to match facts.

This approach that rely on aggregation of facts to employ verification has the advantage that pages are read only once and in any order, instead of many times as in index navigation. The performance is proportional to the number of facts and thus to the number of relationships. The number of facts is about 4 times the number of nodes (in the case where a node is child, a parent and has two neighbors). Since facts can be eliminated as soon as all matching facts are found, there is no relevant impact in storage requirements. Furthermore, the algorithm scales reasonably well yet produces precise error messages.

5.4 Bit Vector Filtering

This approach introduces a further simplification to the aggregation of facts on B-trees with symmetric fence keys. Instead of aggregating facts, the combination of index identifier, page identifies, B-tree level, low and high fence keys is hashed to a value. This value represents the bit position in a bitmap. The bit at this positions is reversed. Since matching facts is based on equality comparisons, facts that match will hash to the same value. Also facts must match in even numbers and as a

consequence the entire bitmap should be back in its original state at the end of the verification.

This approach has the advantage of simplifying the entire operation that was based on sorting and aggregation, allowing a higher performance and less memory consumption. In addition to only being adaptable to B-trees with symmetric fence keys, another drawback of this approach is that this verification only points if the index contains errors or not, not being able to identify the precise type and location of the error.

6 Foster B-Trees

6.1 Motivation

In modern system architectures it is desirable to have data structures optimized for many-core processors as well as for modern memory hierarchies with flash storage and nonvolatile memory. To achieve this, Foster B-trees [GKK12] combine the advantages of B^{link}-trees, write-optimized B-trees and symmetric fence keys to enable simple latching for high concurrency, efficient page migration of nodes to new storage location in flash storage and RAID, wear leveling, efficient defragmentation and continuous and inexpensive self-testing. It also aims at avoiding disadvantages like complex node deletion logic and concurrency control, lack of support for load balancing and structural B-tree changes that require more than two latches. In other words, the Foster B-tree design focus on achieving high concurrency and high update rates without compromising consistency, correctness or read performance.

6.2 Data Structure

In order to combine B^{link}-trees and write-optimized B-trees, Foster B-trees should deal with certain contradictions between these approaches. For example, the design of write-optimized B-trees counts with a single incoming pointer on each node (a parent-to-child pointer) to enable efficient page migration and as a consequence also enable large write operations and efficient defragmentation. However, B^{link}-trees require a link pointer between neighbor nodes in order to implement efficient latching and permit high concurrency in the data structure. The approach taken to deal with these contradictions is to relax certain requirements at an estimated small cost.

Foster B-trees at a stable state look exactly like write-optimized B-trees. However, the new aspect are the intermediate states that may occur during structural changes in the B-tree data structure. Like write-optimized B-trees, at any time each node has only one incoming pointer. However, the requirement of not having pointers among neighbor nodes is relaxed for intermediate states. A link pointer (here called a foster pointer) between neighbor nodes is allowed in these intermediate states. Like B^{link}-trees, only a fixed number of latches is expected during structural changes. Unlike B^{link}-trees, at most two nodes must be latched at any intermediate step for any operation (insert, deletion, load balancing) rather than three nodes in the worst case scenario of B^{link}-trees.

Structural changes can be divided into multiple steps and each step can be a system transaction, since it does not affect the database logical contents. If multiple system transactions are necessary for structural changes, logging optimizations for these transactions are required. Like in write-optimized B-trees, a single log record can capture an entire system transaction while the original source page on the storage device serves as backup until the updated destination page has been written to the storage device.

Finally, in an intermediate state, a Foster B-tree node may be a foster parent. In this case the node has a foster key in additional to its two fence keys. A foster pointer points to the foster child node that contains keys greater than the ones of the foster key in the parent node. In other words, the foster key value separates key values in the foster parent and in the foster child.

6.3 Retrieval

The retrieval of a Foster B-tree node, given a certain key, is very similar to the retrieval in B^{link}-trees. Latch coupling is required for latching in shared mode all

nodes along the path when navigating from node to node in a root-to-leaf traversal. Like in B^{link}-trees, if at some point of the operation the highest key of the current node is lower than the search key, this means that a structural change took place while traversing the tree and the retrieval operation is at the wrong node. In this case, following the foster pointer of the current node is required in order to find the correct node.

6.4 Insertion

The insertion of an entry in a Foster B-tree starts similarly to a retrieval operation, by applying latch coupling with shared latches in a root-to-leaf traversal to locate the leaf node where the entry must be inserted, acquiring an exclusive latch at this leaf node. If there is enough space, the entry is inserted and the operation finishes. However, things get interesting if a node split is required.

Different from B^{link}-trees and other approaches, Foster B-trees split nodes locally without immediate upward propagation. The second step allocates a new node, format it with an empty key range (low and high fence keys have the same value, in this case the foster key) and make this newly allocated node a foster child of the overflowing node. While the new node is not a foster child, it cannot be reached by other threads and hence no latch is required for it up to this point. The third step holds exclusive latches on foster parent (overflowing node) and foster child (empty node) and balance the load between them with appropriate adjustment of the foster key. The final step consists of an adoption, i.e., ending the foster relation. Exclusive latches are acquired for the foster parent and the parent node. The foster key is moved from the foster parent to the parent node as a separating key as well as the foster pointer. The parent node adopts the foster child and becomes its permanent parent.

The intermediate state before an adoption is transient and should be resolved as fast as possible. However, it is still a consistent state and even if other threads observe this state the Foster B-tree is functional. The adoption operation can take place opportunistically as a asynchronous operation during low system load or during a root-to-leave traversal that is able to acquire exclusive latches on the foster parent node and parent node without delay. Since a root-to-leaf traversal may temporarily be longer than minimal during an intermediate state, it is recommended to avoid long chains of foster relations. If this situation arrives, an eager approach can be used to resolve the foster relations by forcing adoption.

6.5 Load Balancing

Load balancing among sibling nodes can be implemented with simple logic similar to the one for the insertion operation. As for insertions, at each step only two nodes are latched on exclusive mode.

In the first step one of the siblings where the load balancing applies and the parent node are latched on exclusive mode. This sibling becomes a foster parent and the other sibling (the one that was not latched) becomes a foster child, i.e., the separating key and the pointer from the parent node is moved into the foster parent node. In the second step both siblings (now foster parent and foster child) are latched on exclusive mode, entries are moved for load balancing and fence keys are adjusted. The third step consists of ending the foster relationship by latching the foster parent node and the parent node and moving the foster key and foster pointer up to the now permanent parent node.

6.6 Node Deletion

In B^{link}-trees we allow a node to be completely empty in order to avoid merging underflowing nodes. However, since Foster B-trees have only a single incoming pointer at all times, the deletion of a node is facilitated and much easier when compared to the deletion in B^{link}-trees. The deletion of nodes might be useful in order to employ free space reclamation.

The steps are similar to the ones followed for a node split, but in an inverse order. In the first step exclusive latches are acquired for the node to be emptied and one of its sibling nodes (neighbor node with the same parent). Foster keys and foster pointers are adjusted in order for the node to be emptied to become foster child and for the sibling node to become foster parent. For the second step, the load from the foster child is moved to the foster parent. If the foster parent has no space to accommodate the load, the node deletion is not possible. At this point, the foster child is completely empty except for the two fence keys. Finally, the third steps consists of removing the foster child from the B-tree and registering it for free space reclamation. Like in the other structural operations, between each step latches are released and re-acquired if necessary, in a way that each step requires only two exclusive latches.

6.7 Structural Verification

As seen in Section 5, symmetric fence keys allow comprehensive verification of B-tree invariants as a side effect of root-to-leaf traversals. At each step of the traversal the child node page identifier, tree level and fence keys can be verified against the facts from the parent node. It is also possible to verify the consistency between neighbor nodes with equality comparisons. Further invariants are verified based on transitive equality. Foster B-trees require additional verification logic to be implemented in order to tolerate and properly verify foster relationships.

Recent work [GKS12] enables Foster B-trees to employ single-page recovery mechanisms based on an auxiliary page recovery index. In this approach the B-tree index is able to self-analyze its own structure for page errors and self=heal from these errors without aborting a transaction or inducing big impact in the system performance.

6.8 Performance Evaluation

Foster B-trees employ a simple structure and simple concurrency control in order to reduce latch contention and consequently improve performance on many-core contexts. A performance evaluation of Foster B-trees [GKK12] shows the following results.

Performance of Selects and Inserts Figure 2 shows the results of experiments using selection queries while varying the number of concurrent threads. Since it is a read-only experiment there is no foster relationship, no logging I/O, no latch conflict and consequently most of the running time is spent on binary searches. The experiment is made in order to isolate the page design of Foster B-trees from the impact of foster relationships. Both original approach (Shore-MT) and the one implemented with Foster B-tree scale well. The difference in the throughput is due to the different prefix truncation algorithms employed. Shore-MT is able to achieve 10% higher compression, but it requires extra effort to reconstruct and compare a single key during binary search. Figure 3 shows the results of experiments using mixed workloads of reads and updates. Shore-MT is compared with Foster B-trees implementing three different policies: no adoption (None), opportunistically

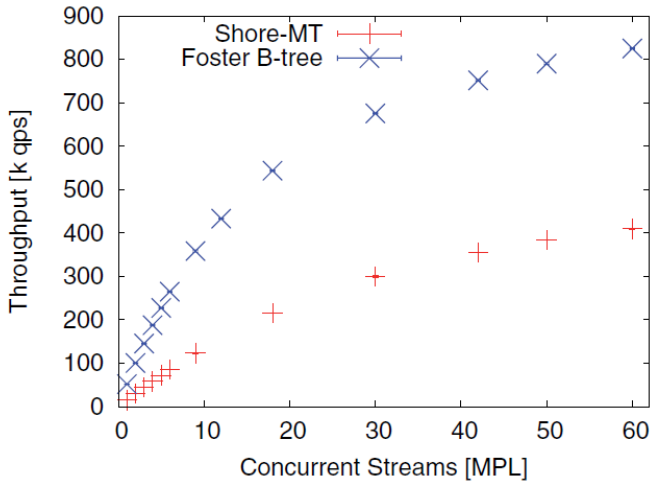

Fig. 2. Read-only workload with varying MPL (MultiProgramming Level).

applying adoption at tree traversals (Opp) and a eager-opportunistically policy that attempts to apply opportunistic adoption but also forces the adoption if it fails repeatedly due to high contention (E-Opp). Foster B-trees perform better due to the page design and to foster relations. Foster relations avoid latch contention and allow Foster B-trees to scale up to 18 threads, where Shore-MT does not scale beyond 9 threads. Since the inserted keys are uniformly random, there are no long chains of foster relationship and thus even the None approach performs well because adoptions are not a must. The bottleneck at the execution with a higher number of threads is directly related to the logging mechanism employed.

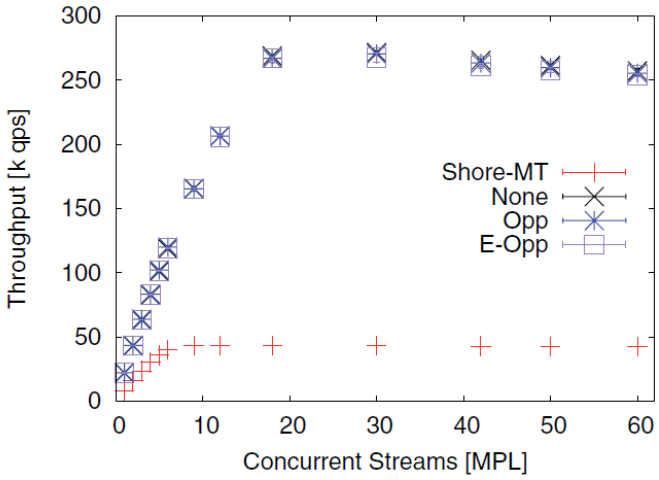

Fig. 3. Insert-only workload with uniform random keys.

Effectiveness of Adoption Policies Figure 4 and Figure 5 show the results of experiments that employ a mixed workload of 80% selection queries and 20% inserts with skewed keys. The goal is to force the creation of foster relations. Figure 4 shows the results for the experiment running in a single thread while the experiments of

Figure 5 run with an increasing number of threads. In Figure 4 it is possible

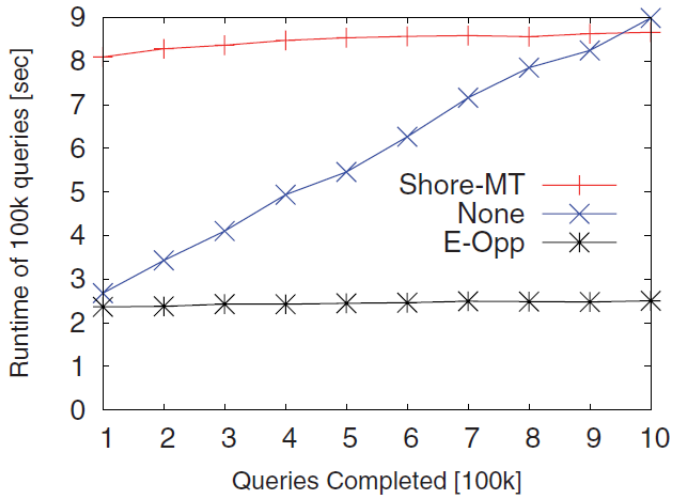

Fig. 4. Time needed to complete groups of 100,000 queries over the lifetime of a mixed, highly-skewed workload.

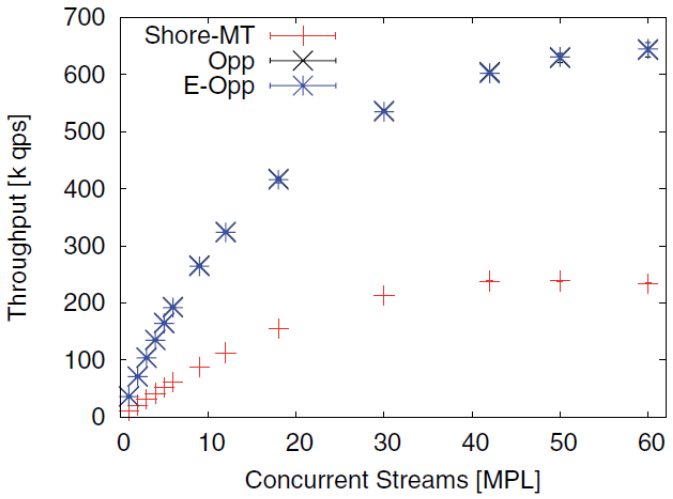

Fig. 5. Select-Insert mixed workload with highly skewed random keys.

to see that E-Opp solves the foster relationships and as a consequence the query runtimes remains practically the same over time. Since None does not resolve foster relationships, the runtime of queries tend to increase with time because of long chain of foster relations. Opp has the same performance of E-Opp and is omitted.

In Figure 5 it is expected that the Foster B-trees perform better than Shore-MT as seen in Figure 2 because of the high number of selections (80%). A small difference between Opp and E-Opp can be perceived with higher concurrency levels and is explained by the rare cases where Opp was not able to solve foster relations quickly enough.

Effects in Cursor Performance Finally, Table 2 shows the experiment employed to compare the differences between employing sibling pointers (Shore-MT) and replacing the sibling pointers with symmetric fence keys (Foster B-trees). The metric used is the time of a single thread to execute ten times a sequential scan of all entries in the B-tree index. Foster B-tree outperform Shore-MT, showing that the negative effect of omitting sibling pointers in detriment of symmetric fence keys is minimal when compared to the gains coming from the new page design.

	Throughput (Records per second)
Shore-MT	40k
Foster B-tree	224k

Table 2. Locks and latches.

7 Conclusion

This work revisited the Foster B-tree data structure. Foster B-trees aim at combining three different B-tree techniques in order to achieve high concurrency and high update rates in a simple and efficient way.

B^{link}-trees are considered for inspiring efficient concurrent manipulation of a database index. The property that any structural operation on the tree structure requires only a small and constant number of latches is highly desirable in order to avoid latch contention. Using a link-pointer between neighbor nodes allows dividing the structural changes in smaller and independent steps. This is the way to achieve the desired behavior.

Write-optimized B-trees, on the other hand, enable large write operations by employing symmetric fence keys and reducing logging effort and log volume. Reducing the costs of a page migration operation is the key. Additionally, the design has to overcome two obstacles in order to succeed where others have failed. First it is required to have a page access performance equal to that of traditional B-trees. Second, the presented design allow a mixture of read-optimized and write-optimized operations that can be managed by any desired policy.

Symmetric fence keys employed by write-optimized B-trees also introduce the advantage of enabling continuous and inexpensive yet comprehensive structural verification of all the B-tree index structure. By solving the cousin problem, it is possible to employ efficient verification algorithms.

Finally, for a Foster B-tree to replicate all these advantages it is necessary to consider and resolve some contradictions between the techniques as well as to avoid disadvantages particular to each of the designs. By behaving most of the time as a write-optimized B-tree, it is required to relax some of the requirements in order to allow intermediate states during structural changes. Consequently, having a single incoming pointer per node at a stable state and dividing structural changes in small steps are the main characteristics of Foster B-tree that enable the mentioned advantages.

References

[BS88] R. Bayer and M. Schkolnick. Readings in database systems. chapter Concurrency of Operations on B-trees, pages 129–139. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988. URL: http://dl.acm.org/citation.cfm?id=48751.48760.

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. Foster b-trees. *ACM Trans. Database Syst.*, 37(3):17:1–17:29, September 2012. URL: http://doi.acm.org/10.1145/2338626.2338630.

[GKS12] Goetz Graefe, Harumi Kuno, and Bernhard Seeger. Self-diagnosing and self-healing indexes. In *Proceedings of the Fifth International Workshop on Testing Database Systems*, DBTest '12, pages 8:1–8:8, New York, NY, USA, 2012. ACM. URL: http://doi.acm.org/10.1145/2304510.2304521.

[GR92] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1992.

[Gra04] Goetz Graefe. Write-optimized b-trees. pages 672–683. VLDB Endowment, 2004.

[GS] Goetz Graefe and Ryan Stonecipher. Efficient verification of btree integrity. *BTW*, pages 27–46.

[LY81] Philip L. Lehman and s. Bing Yao. Efficient locking for concurrent operations on b-trees. *ACM Trans. Database Syst.*, 6(4):650–670, December 1981.