Self-diagnosing and self-healing indexes

Goetz Graefe, Harumi Kuno; Bernhard Seeger
Hewlett-Packard Laboratories; Philipps-Universität Marburg

Abstract

Transactional storage and indexing is the heart of every database, not only for performance and functionality but also for reliability and availability. For high concurrency, these components must be programmed carefully with short critical sections, a variety of consistent states with short transitions, etc. Many-core CPUs exacerbate these requirements. Testing software with 100s or 1,000s of threads is very difficult.

In order to test such code, we suggest verifying the B-tree structure in each traversal. With carefully designed tree structure and node contents, a root-to-leaf pass can verify all nodes along its path comprehensively, i.e., it can verify all B-tree invariants including its consistency constraints with respect to its siblings and cousins (defined below). Thus, instead of testing the index implementation by running a stress-test and verifying the B-tree structure and contents afterwards, i.e., instead of the traditional approach, the test execution itself verifies the structure frequently and efficiently.

During testing prior to a software release, frequent comprehensive verification combined with fast access to all relevant log records permits efficient root cause analysis of test failures. In deployments after software release, frequent verification and fast access to log records permits automatic, reliable, and efficient recovery of the correct, up-to-date page contents. Our contribution is an index structure that gives reliable and efficient access to all relevant log records and thus enables root cause analysis during testing and automatic recovery after deployment.

Categories: H.2. Database management.
General terms: Algorithms.
Keywords: B-tree, quality assurance, failure, recovery.

1 Introduction

The traditional method for testing multi-threaded B-tree code relies on extensive "stress" runs with many concurrent threads, skew in the key value distribution as well as the access pattern, etc. Test success is defined not only by completion of the workload but also by correct final index contents as well as structural consistency of the B-tree. We believe that this is an inefficient way to test multi-threaded code and renders root cause analysis more difficult than necessary.

For example, if a stress test runs for 12 hours and a software defect causes a corruption in an index structure halfway through the test, the test failure will not be known for 6-8 hours, depending on the duration of the database verification. Identifying the root cause requires analyzing hours worth of recovery log. If a page is verified each time it is read from storage into the buffer pool, say once per hour, the corruption will be detected within an hour. Root cause analysis should be much simpler and faster in this case. If, however, each page is verified in each root-to-leaf search of each index structure, and if the verification is comprehensive such that any defect will indeed be found immediately, and if the verification is sufficiently fast that the entire test run is still meaningful, then a defect and its root cause will be identified practically instantly. Figure 1 illustrates the example.

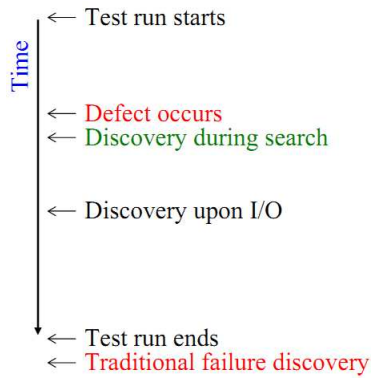

Figure 1. Defect occurrence and discovery.

Optimizing database software for modern processing hardware, in particular for many-core processors, means enabling a large number of software threads in order to exploit all available hardware threads. Testing software with short critical sections in 100s or 1,000s of threads is very difficult. Traditional stress tests run update-intensive application logic in many threads and verify the final database contents as well as its structural consistency. In a B-tree, for example, structural consistency includes pointers and key ranges of related nodes in addition to correct contents within every page.

Optimizing database software for modern storage hardware, in particular RAID devices and flash storage, means enabling large writes, which in turn means moving data pages in each write operation. Thus, data pages move frequently and therefore must move very efficiently. Traditional B-tree variants with forward and backward sibling pointers in addition to parent-to-child pointers are not suitable for frequent page movements, because each page movement requires updates in three pages with pointers.

Moreover, modern disks and their large capacities introduce another issue. For example, scanning a disk of 4 TB at 200 MB/sec takes 20,000 seconds or about 6 hours. Thus, an offline consistency check, even as a disk-order scan (as opposed to an index-order scan), is very disruptive in a production deployment. Online consistency checks are required and incremental consistency checks are extremely desirable.

New B-tree variants are needed to optimize indexing for new hardware and for new usage patterns, e.g., write-

DBTest '12, May 21, 2012 Scottsdale, AZ, U.S.A.

optimized B-trees [G 04]. Development and testing of a new variant of B-trees are very laborious tasks. Our goal is to improve their duration and their outcome by enabling efficient continuous, comprehensive self-testing of B-trees.

B-tree variants differ in their node contents and their invariants. Foster B-trees, for example, eschew forward and backward sibling pointers, leaving only parent-to-child pointers, but including temporary parent relationships among siblings called foster relationships[1]. Moreover, each node contains two fence keys, which define the permissible range of key values in the node. The fence keys in the root node are outside the user's key domain, e.g., $-\infty$ and $+\infty$.

This data structure permits comprehensive testing of all structural invariants in every B-tree access, both read-only queries and updates. Moreover, once all leaf nodes in a key range have been accessed and tested, then all invariants are reliably true. In other words, even if two sibling leaf nodes have the B-tree root as their closest common ancestor, their relationship has been verified, including the requirement that their key ranges immediately abut.

Thus, Foster B-trees enable a new approach to testing multi-threaded indexing software in databases: continuous and online, efficient yet comprehensive, as side effect of queries and updates. What has been missing from Foster B-trees is the capability for self-repair.

If a defect is found during testing, a test can be aborted and the failure analyzed and, hopefully, repaired. If, however, a defect is found in a production system, some form of repair or recovery is required, preferably automatically and immediately. Recent research led to the definition of single-page failures as a new, fourth failure class in databases. The traditional failure classes are transaction failures (e.g., deadlock), media failures (e.g., disk crash), and system failures (e.g., "blue screen"). Recovery of single-page failures can be very efficient if the database includes a page recovery index, a novel data structure designed for the purpose.

Without the page recovery index, single-page recovery cannot efficiently find its start and end points, namely a recent page backup (e.g., a remnant of a page migration) and the most recent relevant log record. The page recovery index speeds single-page repair so much that the user transaction encountering and detecting the failure has no need to abort; it merely experiences a delay of a about a second or less.

Self-diagnosing and self-healing indexes, introduced here, combine the advantages of Foster B-trees and of a page recovery index. They achieve their goals without a separate, dedicated page recovery index, however, by including its information contents within the B-tree index. Thus, a Foster B-tree so modified not only detects any and all faults but also recovers from them without the need for additional data structures other than a traditional recovery log.

In the following section, we describe related prior work including Foster B-trees, single-page failures, and appropriate recovery techniques. Section 3 covers the principal contribution: integration of page recovery information into the B-tree index itself such that no additional data structure is required for recovery and repair after a B-tree node fails a consistency check. In Section 4, we offer a summary and some preliminary conclusions from this research.

[1] A foster parent is "a person who acts as parent and guardian for a child in place of the child's natural parents but without legally adopting the child" [F 12].

2 Related prior work

This section focuses on Foster B-trees and on single-page failures and their recovery. Foster B-trees already have some self-diagnosis functionality and, due to a single pointer per node, enable adding the required information to each pointer that makes the index self-healing. Previously proposed recovery techniques for single-page failures provide the basic mechanisms for self-healing, i.e., a blueprint for the information added to turn Foster B-trees into self-healing indexes. The section provides this background in some detail in order to guide the reader towards the proposed self-diagnosing and self-repairing B-tree variant introduced in the following section.

2.1 Foster B-trees

Foster B-trees [GKK 12] are a new variant of the B-tree data structure. The design goal of Foster B-trees has been 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,
ii. efficient node migration to new storage locations,
iii. and 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 [RO 92]; in database systems, it is known as write-optimized B-trees [G 04]. Both can be exploited not only for large writes but also for wear leveling. In those 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 crucial invariant of Foster B-trees is that for each node, there is a single incoming pointer at all times. This is also the crucial dissimilarity of Foster B-trees and B^{link}-trees [LY 81].

The inexpensive yet continuous and comprehensive verification of all invariants goes beyond error detection and error-correcting codes within each page, which apply to any B-tree or in any other storage structure; it includes all cross-node invariants of the B-tree structure. In other words, all correctness criteria for B-tree structures, including those between sibling nodes (shared parent node), between cousin nodes (shared grandparent node), etc. can be verified reliably in a B-tree with fence keys by simply searching repeatedly and, during all root-to-leaf traversals, comparing values within nodes and between a child node and its parent.

Whether inconsistencies are introduced by defects in the B-tree code, in lower software layers such as replication or file system, or in the hardware (including endurance problems), any violation of any 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 [GK 12].

Foster B-trees forgo sibling pointers in favor of low and high fence keys in each node, which are copies of branch keys posted in a parent during leaf split. Possible concerns about the lack of sibling pointers, e.g., efficiency of scans, cursors, and key range locking, have been addressed earlier [G 04]. Scans with deep read-ahead cannot use the sibling pointers in any case, cursors require additional root-to-leaf B-tree traversals that can be optimized [GKK 12], and key range locking never requires access to another page for a key value to lock if the fence keys themselves can be locked.

Unique among B-tree variants, Foster B-trees permit a node to serve as temporary parent ("foster parent") for its immediate right neighbor (called a "foster child" for the duration of the "foster relationship"). A foster parent carries, in addition to its own fence keys, a foster key that separates its key range from that of its foster child. A foster relationship ends with an "adoption," i.e., the pointer to the foster child is moved (not copied as in a B^{link}-tree [LY 81]) from the foster parent to the permanent parent and the foster key becomes the high fence key in the former foster parent.

Foster relationships are used extensively as intermediate state in changes of the B-tree structure, e.g., during node creation (split), node deletion, and load balancing. A foster child with empty key ranges is used as preliminary state in node creation and as terminal state in node deletion.

Appendix 1 illustrates states and transitions in Foster B-trees. The important characteristic for self-diagnosing and self-healing indexes is the invariant that each node has only a single incoming pointer at all times.

2.2 Single-page failures

[2] For over 30 years, the three traditional failure classes have framed research and development in high availability and reliability of data storage. Extending the set of failure classes requires a strong conceptual motivation as well as a strong case for the practical value of the new failure class and its implementation techniques, including both detection and repair. Earlier work proposed single-page failures as a fourth failure class [GK 12].

All commercial database systems include utilities to verify the integrity of a database, both logical integrity (integrity constraints and materialized views) and physical integrity (pages, indexes, and auxiliary data structures such as allocation bitmaps and compression dictionaries). The relationship between secondary indexes and primary indexes can be part of logical integrity (like a foreign key integrity constraint) or of physical integrity. These tools include Oracle's DBMS_repair and DB Verify, MySQL's myisamchk, IBM's db2dart (database analysis and reporting tool), as well as Sybase's and Microsoft's DBCC (database consistency check). Borisov et al. [BBM 11] introduce a software tool that invokes such tools "proactively and continuously" optimized for least cost and for timely notification of database administrators. Fryer et al. [FSM 12] consider file system errors by continuous testing of individual pages, somewhat similar to the techniques proposed in this paper.

Detection and recovery techniques for single-page failures differ from those for the three traditional failure classes, not only in the algorithms but also in the degree to which recovery must disrupt transaction processing. For example, while media recovery usually takes many minutes or even hours and requires that all affected transactions be aborted, single-page recovery can be designed and implemented such that affected transactions merely wait a short time, perhaps less than a second, for recovery of the lost page contents.

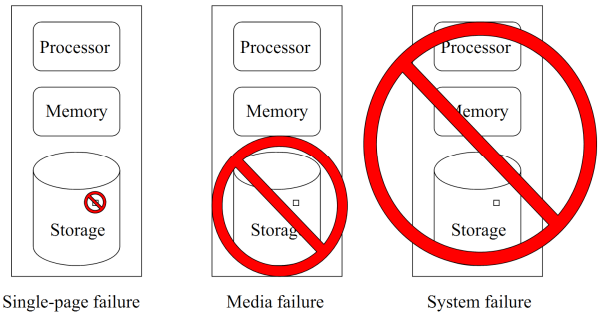

Figure 2. Failure scopes and possible escalation.

Figure 2 illustrates possible failure scopes. If single-page failures are not a supported class of failures, failure of a single page (left) must be handled as a media failure (center). In machines or nodes with only one storage device, a media failure is equal to a system failure (right). Single-page failures as a new failure class prevent such an escalation.

Central to the proposed recovery techniques [GK 12] is a new data structure, one instance per database, called the page recovery index. It maps each database page to its most recent log record and its most recent backup page. The backup page might be part of a database backup or retained after a page migration, among other possible sources of page backups. The PageLSN, i.e., the log sequence number of the most recent log record applied to a data page [MHL 92], is copied from the data page to the page recovery index after the data page is written back to the database. The required effort is similar to that of logging completed writes, a technique already employed in some database systems in order to speed up recovery from a possible system failure.

Appendix 2 illustrates data structures and logical flow of single-page recovery. Recovery of a single data page can be sufficiently fast that there is no need to abort the affected transaction, even if it experiences a short delay.

3 Self-repairing B-trees

A self-repairing B-tree combines the facilities of Foster B-tree and page recovery index into a single structure. In

[2] This section is adapted from [GK 12].

other words, the central advancement from the original Foster B-tree to a self-repairing B-tree is to move the information from the page recovery index into each index itself. Self-repairing B-trees move the auxiliary information required for single-page recovery into the B-tree itself.

When a page recovery index exists, it carries, for each data page in the database, the location of the most recent page backup and of the most recent log record. The most recent backup may be the old image during a page movement, e.g., during defragmentation or during write-optimized relocation, a format record, a full database backup, or an incremental database backup. The most recent log record is indicated by a copy of the PageLSN [MHL 92] on the database page. While a page is present in the buffer pool, this information need not be updated; whenever the database page is written to the database, the page recovery index keeps track of the PageLSN most recently written to storage.

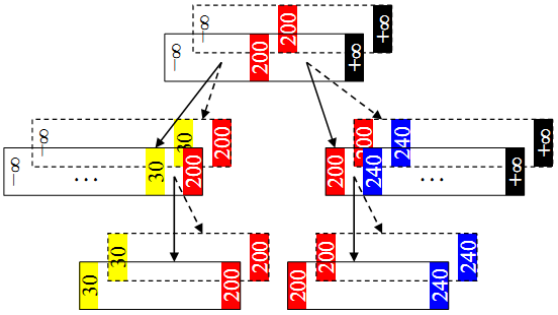

Figure 3. Pointers to backup pages in a self-repairing B-tree.

Figure 3 illustrates the B-tree of Figure 7 as a self-repairing B-tree. Backup pages and pointers to backup pages are included and drawn with dashed lines. Pointers into the recovery log, i.e., log sequence numbers of the child pages, are omitted. The backup pages may part of a full database backup or in a differential backup (i.e., only recently changed pages), found in the recovery log as a full-page copy or as a formatting record, or remain from a page migration during defragmentation or garbage collection.

The page recovery index pertains to any storage structure, not just to B-tree indexes. In B-tree indexes with only a single incoming pointer per node, e.g., write-optimized B-trees or Foster B-trees, this pointer may be augmented by the information of the page recovery index. This additional information turns a write-optimized B-tree or a Foster B-tree into a self-repairing B-tree.

For pages of a self-repairing B-tree within a database with a page recovery index, there are no entries in the page recovery index. Each entry in a branch node of a B-tree carries not only a key value and a child pointer but also pointers to the most recent page backup and the most recent log record. As in a page recovery index, these fields are updated whenever the child page is formatted or moved or written to storage after an update.

In a database with self-repairing B-trees but no other storage structures, there is no need for a page recovery index. Instead, single-page recovery is enabled only for those self-repairing B-trees. For failures or inconsistencies in all other pages, mechanisms and policies for media recovery apply.

3.1 Self-diagnosis

Self-diagnosis of a storage structure and its code offers great value both prior to release, i.e., during quality assurance, and after release, i.e., in production deployments. The proposed self-repairing B-tree, however, provides value beyond self-testing. By making the history of a database page immediately available in form of a pointer into the recovery log, it enables efficient root cause analysis of defects.

In order to maximize this value, two additional small modifications of the B-tree code recommend themselves. First, the indexing code should frequently invoke the consistency checks ordinarily invoked only after reading a page from storage into the buffer pool. Second, successful verification of each B-tree page should be recorded.

Frequent invocations of the verification routine ensures that defects are found quickly after defective code has been executed. For example, comparison of fence keys in a child node with the branch keys in its parent can easily be invoked in every root-to-leaf pass. More exhaustive checks, e.g., the sort order of key values or space allocation within a node are fairly expensive and should be used judiciously. Fortunately, changes within each node usually occur with an exclusive latch on the entire page (in the buffer pool). Inconsistencies between nodes, on the other hand, are more likely to be caused by defects in the concurrency control code, in particular excessive "performance optimization." Thus, verifying the cross-node invariants frequently and cheaply is important.

Successful verifications should be recorded, possibly in the recovery log but more likely in the buffer pool descriptors of pages. In the latter case, inclusion of a PageLSN links the last successful verification to a log record. Thus, a failed test must have been caused by a change logged thereafter. Frequent verification together with immediate access to the relevant log records promises focused and efficient root cause analysis of defects in multi-threaded B-tree code.

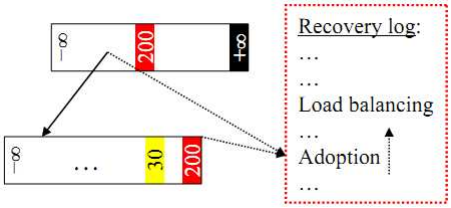

Figure 4. Pointers to log records in a self-repairing B-tree.

Figure 4 illustrates the root node and the left branch node of the Foster B-tree in Figure 7 as a self-repairing B-tree. Backup pages and their pointers are omitted; dotted lines show the recovery log and pointers to specific log records. The log sequence number information associated with the child pointer in the root node points to the same log record as the branch node with its PageLSN in its page header. Thus, a root-to-leaf traversal in a self-repairing B-tree can efficiently verify one more fact than a Foster B-tree.

All page accesses in all storage structures can verify page-internal variants, write-optimized B-trees and Foster B-trees can comprehensively verify structural invariants of the data structure, and self-repairing B-trees can also verify the PageLSN in a child page by comparing it with the information in the parent page.

In the case shown in Figure 4, the most recent update of the branch node is the adoption, i.e., moving branch key value 30 and the pointer to the newly allocated page from the foster parent to the permanent parent. The adoption log record points, using a log sequence number, to the previous log record pertaining to the same page. Per-page log chains, e.g., the pointer between two log records, are required for efficient single-page recovery, described in more detail elsewhere [GK 12], and already present in various commercial database management systems.

3.2 Self-healing

After deployment, frequent verification of index consistency prevents cascading damage and permits timely repair. While repairing a single database page can be a tedious task with today's software tools, unwinding ensuing damage, e.g., in secondary indexes and materialized views, is frightful.

The proposed data structure permits automatic and very efficient repair. This capability equals that of a database with a page recovery index; the difference is that a self-repairing B-tree encapsulates all information in a single data structure.

After a repair is complete, another invocation of the verification logic can verify its success. Thus, the proposed self-repairing B-tree verifies not only normal operations but also its own repair tasks.

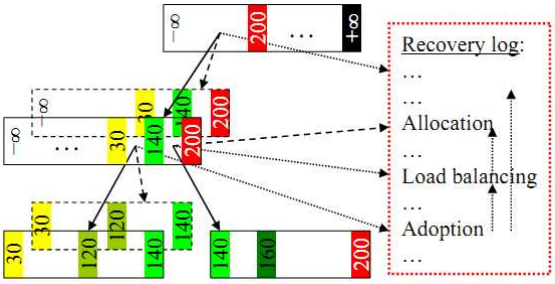

Figure 5. Nodes and pointers in a self-repairing B-tree.

Figure 5 illustrates the relevant parts of the Foster B-tree in Figure 10 as a self-repairing B-tree. Backup information is drawn as dashed lines; log sequence numbers and the recovery log are shown with dotted lines. The backup page of the root node is omitted from the diagram. A backup page for the newly allocated node does not exist yet; instead, the log record describing allocation and initial formatting of the page serves the purpose. Using this log record, a backup page can be created and formatted if required; thus, this log record is as good as a backup page.

For example, if a search for value 160 finds the leaf page unreadable or corrupted, it can repeat the entire history from page allocation and formatting through load balancing. Note that adoption, i.e., moving branch key and pointer from foster parent to permanent parent, does not affect the former foster child. Thus, the branch entry for the former foster parent points to the log record for the adoption while the branch entry for the former foster child points to the log record for load balancing between foster parent and foster child.

Figure 5 shows a log record with two pointers to prior log records. This is necessary since an adoption affects two pages, the foster parent and the permanent parent. If logging is optimized such that the updates of both pages are captured in a single log record, this log record must participate in two per-page log chains. Moreover, the diagram shows the root node still pointing to an earlier log record for its left child, i.e., before the appropriate update. This update may be delayed until the page has been written from the buffer pool to permanent storage [GK 12]. In the meantime, comprehensive self-testing may need to follow a few steps in a per-page log chain. With recent log records still in memory, all required information remains readily available for root cause analysis during quality assurance and for efficient recovery in production systems.

The possible sources of a backup page include database backups, log records (e.g., allocation and formatting of a new page), remnants of a page migration (e.g., in write-optimized B-trees on RAID or flash devices), or page copies taken specifically as backup pages. When a new page backup is created, the prior one is obsolete and can be released. In the normal case, the up-to-date copy of the data page is accessible and usable; thus, it is sufficient if each data page points to its own backup. In the case of a single-page failure, i.e., cases in which the up-to-date copy of the data page is not usable and single-page recovery is required, scanning the per-page chain of log records will lead to (and end) at the log record describing formatting, migration, or taking a backup copy. For example, Figure 5 shows a per-page chain of log records ending at the allocation log record.

Therefore, it is not truly required that a parent include with each child pointer a pointer to the child's backup page as shown in Figure 3 and in Figure 5. Instead, a self-repairing B-tree actually requires only the log sequence number pointing to the most recent log record. This log sequence number is required for efficient verification of a parent-to-child relationship and of the child page during a search; a pointer to the child's backup page is not really required during the search and the verification. In other words, the data structure shown in Figure 4 more closely resembles the required data structure. Incidentally, this modification also solves a problem glanced over in Figure 5, i.e., where to keep a pointer to a backup page of the B-tree's root node.

Figure 6 illustrates this modification applied to the B-tree of Figure 5. The parent nodes and their child pointers omit pointers to backup pages; instead, backup pages are referenced in the up-to-date pages and in the recovery log. For the recently allocated leaf page, there is no backup page yet; therefore, the per-page chain in the recovery log ends in a log record for allocation and formatting, not for page migration or taking a page backup.

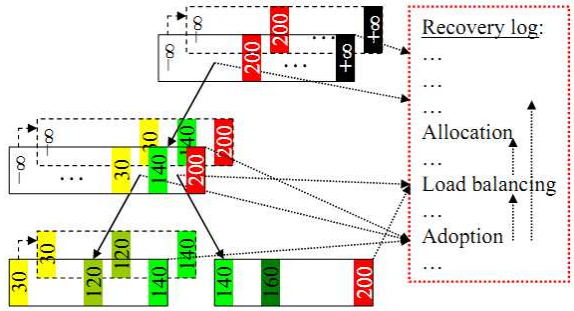

Figure 6. Self-repairing B-tree with local backup pointers.

Finally, this modification applies not only to self-repairing B-tree indexes but also to single-page recovery using a page recovery index [GK 12]: the page recovery index only requires a log sequence number pointing to the most recent log record but not a pointer to a page backup.

4 Summary and conclusions

In summary, self-repairing B-trees combine techniques from Foster B-trees and single-page recovery. The result is a self-diagnosing and self-healing index structure.

Figure 9 and Figure 10 epitomize Foster B-trees: root-to-leaf passes of uniform length in the steady state, symmetric fence keys in every node, a single pointer to each 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 small transacted steps. Small transactions resonate with many-core CPUs, local overflow further reduces concurrency contention, and a single pointer permits easy page movement and thus forming large writes best for RAID and flash devices.

Figure 2 illustrates single-page failures and Figure 12 their recovery using a page recovery index that maps each database page to its most recent log record and backup page. While even traditional disk drives suffer single-page failures, databases on modern storage hardware such as flash memory and non-volatile memory will benefit even more from efficient detection and recovery of single-page failures.

Figure 5 and Figure 6 show how the proposed self-repairing B-tree can embed this information in the index structure itself, superseding a separate page recovery index. Frequent B-tree consistency checks and recording of successful checks permit efficient root cause analysis during quality assurance and prevents cascading damage after deployment.

In conclusion, self-repairing B-trees speed up testing of a core component of practically every data storage system. Thus, they enable more rapid development as well as more confidence in the released software, its quality assurance, and its reliability in production deployments.

By adding autonomic fault detection and recovery to B-trees, their advantages are immediately available to all kinds of indexes implemented as B-trees with specially constructed keys, including hash indexes (as B-tree on hash values, perhaps with large pages and interpolation search), multi-dimensional indexing (using B-trees on space-filling curves, e.g., the Z-order UB-tree), master-detail clustering (by merging multiple indexes into a single B-tree), column stores (implemented as B-trees on row identifiers, with suitable run-length encoding in the leaves), very large field values (also known as blobs, implemented with B-trees on byte counters), and time travel (by appending version identifiers to user-defined B-tree keys, plus appropriate compression).

Acknowledgements

Hideaki Kimura participated in the design of Foster B-trees and provided a first implementation as well as an evaluation that demonstrated their superior performance and scalability compared to the native indexes of Shore-MT.

References

[BBM 11] Nedyalko Borisov, Shivnath Babu, Nagapramod Mandagere, Sandeep Uttamchandani: Warding off the dangers of data corruption with Amulet. ACM SIGMOD 2011: 277-288.

[G 04] Goetz Graefe: Write-optimized B-trees. VLDB 2004: 672-683.

[G 10] Goetz Graefe: A survey of B-tree locking techniques. ACM TODS 35(3) (2010).

[G 12] Goetz Graefe: A survey of B-tree logging and recovery techniques. ACM TODS 37(1):1 (2012).

[GK 12] Goetz Graefe, Harumi A. Kuno: Definition, detection, and recovery of single-page failures, a fourth class of database failures. PVLDB 5(7):646-655 (2012).

[GKK 12] Goetz Graefe, Hideaki Kimura, Harumi Kuno: Foster B-trees. Submitted for publication (2012).

[GS 09] Goetz Graefe, Ryan Stonecipher: Efficient verification of B-tree integrity. BTW 2009: 27-46.

[F 12] http://www.thefreedictionary.com/Foster+parent, retrieved April 13, 2012.

[FSM 12] Daniel Fryer, Kuei Sun, Rahat Mahmood, TingHao Cheng, Shaun Benjamin, Ashvin Goel, Angela Demke Brown: Recon: verifying file system consistency at runtime. USENIX FAST 2012: 73-86.

[L 04] David B. Lomet: Simple, robust and highly concurrent B-trees with node deletion. ICDE 2004: 18-27.

[LY 81] Philip L. Lehman, S. Bing Yao: Efficient locking for concurrent operations on B-trees. ACM TODS 6(4): 650-670 (1981).

[M 90] C. Mohan: ARIES/KVL: a key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. VLDB 1990: 392-405.

[MHL 92] C. Mohan, Donald J. Haderle, Bruce G. Lindsay, Hamid Pirahesh, Peter M. Schwarz: ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. ACM TODS 17(1): 94-162 (1992).

[ML 92] C. Mohan, Frank E. Levine: ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. ACM SIGMOD 1992: 371-380.

[RO 92] Mendel Rosenblum, John K. Ousterhout: The design and implementation of a log-structured file system. ACM TOCS 10(1): 26-52 (1992).

Appendix 1 Foster B-trees

We include here some detailed example figures because Foster B-trees not yet published.

Figure 7 shows a Foster B-tree with parent-to-child pointers, symmetric fence keys in every node, and fence key values −∞ and +∞ along the left and right edges of the tree. The fence keys along a "seam," e.g., from the branch key value 200 in the root node to the leaf level, enable consistency checks in every root-to-leaf pass during search and update operations. They also enable comprehensive consistency checks for the entire B-tree using a disk-order scan (as opposed to an index-order scan) and a bitmap [GS 09]. The next diagrams show steps in splitting an overflowing node.

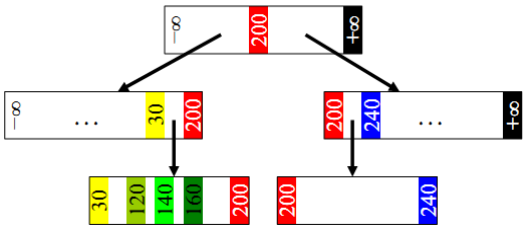

Figure 7. A Foster B-tree prior to a leaf split.

Figure 8 shows the state of the B-tree immediately after allocation of a page to serve as a new leaf node. At this point, the new page has been formatted as an empty page with an empty key range. Low and high fence keys are equal. In fact, in a concrete implementation, there may be only a single record in the page that serves as both left and right fence keys. Since one fence key is an exclusive bound, two equal fence keys imply an empty key range. Similarly, the overflowing node requires only one copy of its high fence key. Thus, the left leaf in Figure 8 requires only one additional pointer (a page identifier) for the new page compared to the state shown in Figure 7. The next step is load balancing among the foster parent and the foster child.

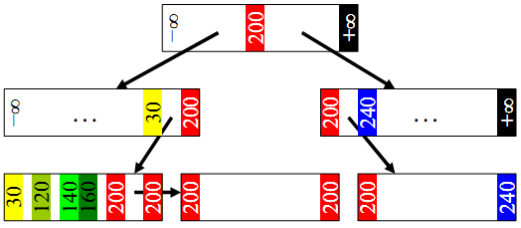

Figure 8. Preliminary state during leaf split.

Figure 9 shows the intermediate state after load balancing among foster parent and foster child. Note the key value 140 as foster key in the foster parent and as low fence key in the foster child. Since the foster parent (and not the permanent parent) carries the branch key that guides searches and updates to either the foster parent or the foster child, the permanent parent is not involved in load balancing among sibling and replacement of the branch key value.

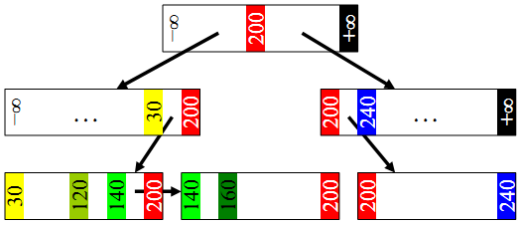

Figure 9. Intermediate state during leaf split.

Figure 10 shows the Foster B-tree after the split is complete, i.e., after adoption of the foster child by the permanent parent. Each of these state transitions is a system transaction with a commit record in the recovery log. These states are visible to other transactions as well as to consistency checks. For example, in the preliminary state shown in Figure 8, the newly allocated page is part of the index and its entry in the data structures for free space management is committed and released (no longer locked or latched).

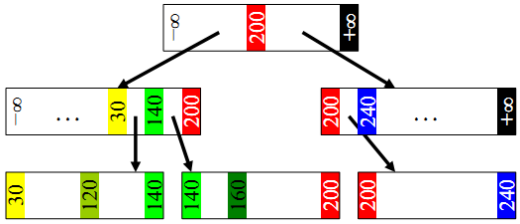

Figure 10. Final state after a leaf split.

For node deletion, the sequence of steps is opposite, except that the state shown in Figure 8 may be called a terminal state (instead of preliminary state) just prior to releasing the page to free space management. The Foster B-tree is unique among B-tree variants as no action latches more than two nodes at a time, even including node deletion, for which other designs require a "tree latch" [L 04, M 90, ML 92]. Moreover, the guarantee that each node has exactly one incoming pointer at a time permits efficient migration during defragmentation, garbage collection, and write-optimized operation on RAID and flash devices. Each of these small steps with their characteristically short critical sections should be implemented as a system transaction [G 10, G 12].

Appendix 2 Single-page recovery

Single-page failures and the page recovery index are the foundation of the design proposed here.

Figure 11 illustrates the logic for reading a page after a buffer fault. If a newly read data page fails the appropriate consistency check, a traditional system must declare a media failure. If single-page failures are supported, the system keeps running using a single-page recovery procedure.

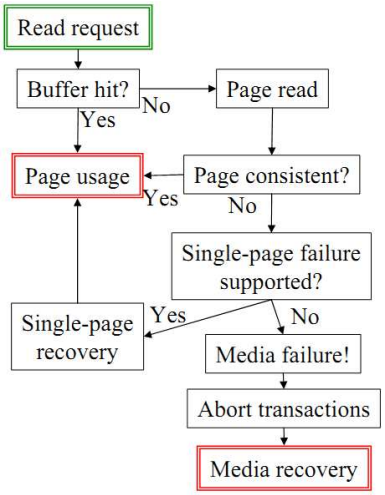

Figure 11. Page retrieval logic.

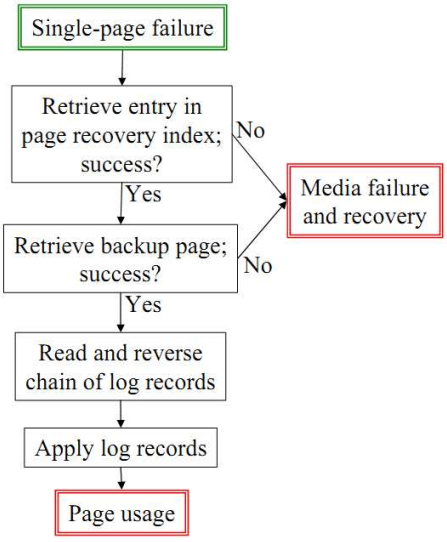

Figure 12. Single-page recovery logic.

Figure 12 illustrates the steps in a single-page recovery. If anything fails, e.g., retrieval of an appropriate entry in the page recovery index, the system can resort to a media failure and appropriate recovery, of course with the implied disruption of transaction processing. More often, single-page recovery can be expected to succeed using a page backup and log records linked together in the per-page log chain. Per-page log chains already exist in many database systems, for example in Microsoft SQL Server.

Once the page contents have been recovered and brought up-to-date in the buffer pool, the page can be moved to a new location. The old, failed location can be released to the free space pool or registered in a bad block list. Obviously, differently from the process in other page migrations, the failed page must not be recorded as a backup page in the page recovery index.

Appendix 3 Efficient maintenance

In the original design for single-page failures [GK 12], the page recovery index is updated only after writing a child page. Thus, while a copy of a data page remains in the buffer pool, the page recovery index may fall behind. This update requires one log record; a separate commit log record can be omitted or integrated into the update log record. Writing a single log record after successfully writing a data page back to disk is acceptable overhead; in fact, it already exists in those systems that log successful writes in order to speed up recovery from system failures by avoiding most unnecessary I/Os during recovery.

We propose to update parent pages in a similar way, i.e., only after writing the child page back to disk. In other words, a parent page reflects the PageLSN of the child page on disk, not in the buffer pool. Thus, in a parent-to-child step during a root-to-leaf traversal, the log sequence number in the parent and in the child may differ.

Appendix 4 Verification

In the original design for single-page failures, the log sequence number in the page recovery index is used to verify the log sequence number in the data page only after a page is read into the buffer pool. At these times, the log sequence number in the page recovery index and in the newly read page must be equal.

If the log sequence number in a data page is tested during each root-to-leaf traversal of a self-diagnosing B-tree index, the values may differ. In this case, the log sequence number in the child must be later and it must, via the per-page log chain, lead to the log record indicated by the log sequence number in the parent. While not an immediate equality test, it can be performed very efficiently if the tail of the recovery log remains available in the buffer. Note that older log records are a read-only data structure and thus require no concurrency control (latching).