An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System

Mark Sullivan *Michael Olson*

Department of Electrical Engineering and Computer Sciences
University of California
Berkeley, California 94720

Abstract

This paper presents two algorithms for maintaining B-tree index consistency in a DBMS which does not use write-ahead logging (WAL). One algorithm is similar to shadow paging, but improves performance by integrating shadow meta-data with index meta-data. The other algorithm uses a two-phase page reorganization scheme to reduce the space overhead caused by shadow paging. Although designed for the POSTGRES storage system, these algorithms would also be useful in a WAL-based storage system as support for logical logging. Measurements of a prototype implementation and estimates of the effect of the algorithms on large trees show that they will have little impact on data manager performance.

1. Introduction

The POSTGRES storage system uses no-overwrite techniques to combine support for historical data with support for transaction management [13]. Instead of write-ahead log processing, POSTGRES recovers from failures by falling back to the latest version of its preserved historical data. Using historical data in place of a conventional log gives POSTGRES important availability and reliability advantages over other database management systems. Data availability improves because the DBMS can restart after a failure in seconds. The database is always consistent without log processing, so restart need only initialize in-memory data structures. By eliminating log processing, POSTGRES provides transaction support without special recovery code. Avoiding complex recovery code increases DBMS reliability.

The original POSTGRES storage system as described in [13] treated indices and heap (unkeyed) relations in the same manner. [13] describes data structures used to determine which tuples were created by transactions which never committed. The DBMS ignores these tuples after a failure. However, because indices often have higher concurrency requirements than heap relations, most data managers must treat the two differently. For high concurrency, POSTGRES indices must use non-two-phase locks which do not work well with the original POSTGRES storage system.

A more important problem for index management is that index data structures include pointers between disk pages. A single update to the index can change several pages and the pointer links among them. Failing after some but not all of the pages have been written to stable storage leaves the index inconsistent. In a DBMS which uses a write-ahead log (WAL) protocol for recovery, the atomicity of index updates is guaranteed by log processing at recovery time. POSTGRES has no log, so it requires other solutions.

In [10], the DBMS maintains consistency of B-tree indices by adding extra synchronous disk writes and by controlling page write order. POSTGRES index management assumes that synchronous writes to a single file are *unordered* for two reasons. First, using several synchronous writes per page split would significantly worsen page split performance. Controlling write order in a single (multi-page) synchronous write is impossible in UNIX-based operating systems and would worsen the performance of disk scheduling algorithms even if it were possible. A second and more important reason not to depend on write ordering for index management is that it will not work for some common kinds of indices. The B^{link}-trees used in POSTGRES have several paths to any B-tree leaf page. No write order sequence exists that will leave the data structure consistent during the entire page split (example in Section 3.5).

This paper presents two general techniques for maintaining index consistency without using write-ahead logging. Although we have implemented them only for B^{link}-trees, the same techniques can be used for R-trees [6], extensible hash indices [4], and other B-tree variants such as B^{*}-trees [3]. In both techniques, the DBMS detects on first use any inconsistencies in the index caused by interrupted updates. When an inconsistency in the index is discovered, consistency is restored by reexecuting incomplete page split or merge operations. Again,

This work is supported by NSF grant MIP-8715235.

to maintain reliability, the two techniques largely avoid special case recovery code. The recovery operation for a page split is nearly the same as the normal page split operation.

One of the two techniques uses a no-overwrite strategy which is similar to shadow paging [9]. The before image of a page to be split is left intact on stable storage until the two half-pages resulting from the split have been written out. Although recovery mechanisms based on shadow paging have been abandoned in commercial systems because of the performance problems experienced by System R [5], they are a practical mechanism for managing indices. Shadow paging makes sequentially ordered pages in the file non-sequential on the disk. While non-sequential ordering ruins the performance of clustered relation scans, it is not an issue for index files. The shadow paging technique, however, still has larger space overhead than a normal index.

The second technique, page reorganization, eliminates that space overhead, but performs poorly when the same index page splits many times during the same transaction. The page reorganization scheme ensures that keys moved from one page to another in a split are always available on either the source or destination page. A hybrid between the two algorithms could preserve the best features of each. Using shadow paging near the leaf pages where splits are most common would improve split performance; using page reorganization nearer the root would reduce space overhead.

The index management techniques used in POSTGRES can even improve the performance and reliability of a conventional WAL DBMS which uses logical logging to record index updates. B-tree index implementations often require physical logging of the keys involved in page splits or merges in order to maintain consistency (e.g. [11]). Combining logical logging and the POSTGRES shadow paging or page reorganization indices would make the write-ahead log more compact and prevent B-tree keys corrupted by software errors from propagating into the log.

This paper is divided into five parts. The first and second list assumptions and describe the new index management techniques. A third section discusses the implications of the technique for logical logging in a WAL storage system. The fourth and fifth sections model and measure performance impact.

2. Assumptions

As in [8], we assume that no duplicate keys are stored in indices. In POSTGRES, duplicate keys are turned into <*value*, *object_id*> keys which are guaranteed to be unique before they are entered into the index.

In POSTGRES, all pages touched by a transaction must be written to stable storage before the transaction commits. For the purposes of this paper, when the DBMS **syncs** its pages, all modified pages are written to disk. They are written to disk in an order chosen by the operating system, not the DBMS. When a crash occurs during a sync operation, any subset of the synced pages may have been written to disk. We assume that single-page disk writes are atomic. The sync system call is assumed either to block the DBMS or to notify the DBMS when all the page writes have been completed. The sync operation corresponds to the support for write ordering provided by the UNIX operating system.

To make the index recoverable without log processing, the DBMS must ensure that currently valid keys are visible and invalid keys are invisible to index lookup operations. The POSTGRES storage system can detect and ignore records pointed to by invalid keys, so recovery only needs to ensure that valid keys are not lost.

In POSTGRES indices, there are two possible sources of inconsistencies: *inter-page* and *intra-page* inconsistencies. Inter-page inconsistencies occur when a pointer to page *B* is stored in page *A*. A failure could occur after *A* has been written to stable storage but before *B* has been. An intra-page inconsistency happens if a page is written to stable storage while the DBMS is adding a key to the page. This can happen easily in POSTGRES if two transactions insert keys into the same page. If the first commits and forces the page to be written to stable storage while the second is in the middle of an insert, the page on stable storage will be inconsistent. After a crash, the DBMS must be able to detect that the page is inconsistent and repair it.

3. Support for POSTGRES Indices

3.1. Traditional B-tree Data Structure

In a traditional B-tree [1], each page of the tree contains an array of <*key*, *data*> pairs and a header which describes space allocation on the page. The order of the keys on the page is recorded by a **line table** (described in [11]). Each entry of the line table contains an offset to the beginning of a <*key*, *data*> pair in the page. If a new key is added to a page, the line table entries are reordered, not the <*key*, *data*> elements stored on the page. On an *internal page*, the data element associated with a key is a pointer to a child page. On a *leaf page*, the data element associated with a key is a tuple identifier (TID) – a pointer to a data page and a line table entry on that page.

Comer [3] describes B-tree data structures in some detail, so we will not describe the basic page split and merge operations. Lanin and Shasha [7] show that B-tree merge operations can be handled using an algorithm analogous to the page split algorithm. Their observation about traditional B-trees is also true for the new B-tree data structures described in this paper, so we will focus on B-tree splits in the discussion that follows.

3.2. Sync Tokens and Synchronous Writes

The POSTGRES index management algorithms use a global **sync counter** maintained by the DBMS to remember which pages were written out during a given sync operation. After every sync operation in which an index split occurred, the DBMS increments the global sync counter. A **maximum sync counter** guaranteed to be larger than the global sync counter is maintained on stable storage. If the current global sync counter approaches the maximum, a new maximum must be chosen and written to stable storage. After a crash, the maximum sync counter is used to reinitialize the global sync counter.

A **sync token** is the value of the global sync counter at one point in time. Sync tokens are saved on index pages to detect inter-page inconsistencies. The **last crash sync token** is the initialization value used when the DBMS recovered from the most recent system crash. If the DBMS shuts down cleanly, the global sync counter and last crash sync token are written to stable storage.

3.3. Technique One: Shadow Page Indices

In POSTGRES shadow B-trees, every key on an internal page contains a pointer to the current and previous version of the child page associated with the key. Instead of an array of <*key*, *childPtr*> pairs on the page, the shadow B-tree page is an array of <*key*, *childPtr*, *prevPtr*> triples (see Figure 1). The previous page associated with a key is a page containing the key value which is guaranteed to be on stable storage. If the childPtr is ever found to be inconsistent, the prev page is used to build a new child page.

When splitting a B-tree page, P, two new pages are allocated − call them P_a and P_b. Half of the keys from P are copied to P_a and half to P_b. During the split, the keys on P are neither modified nor overwritten. When P_a and P_b are initialized, the value of the global sync counter is recorded in a **syncToken** field in each page's header.

After the split, P's parent page, A, must be updated. Page A initially contains a key K1 which points to P. The traditional B-tree split algorithm calls for a new key, K2, containing a pointer to P_b, to be added to A. In the shadow paging algorithm, A is updated in the following manner:

(1) The new key K2 is allocated on A. K2's childPtr field contains the page number of page P_b.

(2) If P's sync token is different from the current global sync counter, P must have been written to stable storage already. In this case, the prevPtrs for both K2 and K1 are set to point to P, and P is added to an in-memory to-be-freed list. After the next sync operation, P will be added to the index freelist.

(3) If P's sync token is the same as the current global sync counter, the prevPtr for K1 must be reused since P is not yet on stable storage. K1's prevPtr is assigned to K2's, and P is freed immediately. This situation only occurs if two splits occur at the same key between sync operations.

(4) K2 is inserted into the page A's line table.

(5) Key K1 is modified so that its childPtr field contains the page number of P_a instead of P.

If adding K2 to the page A causes A to split, the same algorithm is followed unless A is the B-tree root page. If the root page splits, a new root page is created containing two <*key*, *data*> pairs pointing to the two halves of the old root. The first page of the index is a meta-data page containing a pointer to the current root of the tree. Like internal page keys, the root pointer must contain a previous and current page pointer.

In order to prevent an intra-page inconsistency, we must be careful when adding K2 to the line table. The line table entries are intra-page pointers − offsets within the page − which point to key values. The line table is ordered, so the line table entry following K1's offset is selected to hold K2's offset. The line table is extended by first copying the last entry in the line table one element beyond the line table, then incrementing the **nKeys** field of the page header. Next, all of the line table entries between K1's and the last one are copied one entry to the right of their current position. Finally, K2's offset is saved in the entry after K1's.

3.3.1. Detecting Inconsistencies in the Index

A crash during a B-tree update can cause an inconsistency only if the parent, A, is written to stable storage before the crash, but not the child. In that case, A points to an uninitialized page or a page that has been reused. If A was not written, then the new child page is inaccessible, but the parent-child link is consistent.

When descending from A to P, the DBMS determines from A the minimum and maximum key values that should be on P before stepping from A to P. At P, the minimum and maximum key values actually present

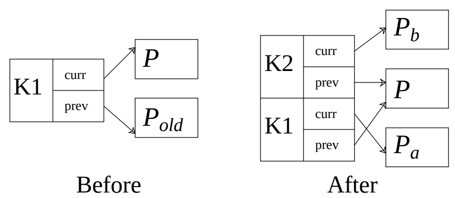

Figure 1: Shadowing Page Split

on the page are compared to the expected key range. If the key ranges are the same, the parent-child link is consistent and the search can continue. If the key ranges differ or if the page is zeroed, the DBMS has detected an inter-page inconsistency. Intra-page inconsistencies are detected when two adjacent entries in the line table contain the same offset value.

3.3.2. Repairing Inconsistencies in the Index

As soon as a broken inter-page pointer link is discovered, the DBMS must redo the interrupted page split operation. The prevPtr shows the page that existed before the split. To reinitialize the out-of-date child page, the DBMS uses the keys on the parent page to determine the range of keys that were on the missing page. These keys are copied directly to the child page from the page pointed to by prevPtr. The sync token on the child page is initialized to the current global sync counter. After the child page has been reinitialized, the B-tree search can continue.

If the root page is split and the new version of the root is lost, the prevChild page is copied directly to the child page. If no root page existed before the failure (i.e. all keys inserted into the tree were lost), the root has no prevChild page and is initialized to an empty page.

The DBMS repairs an intra-page inconsistency by deleting the duplicate entry. The DBMS copies line table entries left until the duplicate is the last entry in the line table, then, decrements nKeys in the page header.

3.3.3. Free Space Management

During normal operation, pages freed from an index are kept on an in-memory freelist associated with that index. Because the freelist is in volatile storage, it does not survive system failures and must eventually be regenerated after a failure. In a UNIX-based file system, a new page may always be allocated, when the freelist is empty, by extending the index file.

POSTGRES heap relations require a garbage collector as part of the storage system's archiving feature [13]. Adding index freelist regeneration to its current archiving tasks does not make garbage collection much more expensive. If the DBMS is shutdown cleanly, the current index freelist should be written to disk. When the DBMS is restarted, the freelist on disk must be deleted before any of the pages on the list are reallocated. Otherwise, a crash will cause the old freelist to be valid again and allow the pages to be allocated twice.

For shadow indices, the key range associated with each page in the freelist must be stored in the freelist along with the page number. Key ranges are used to detect inconsistencies that occur when the child page was not written to disk. If the same page were reallocated for the same key range, there would be no way to tell if the new version of the page were lost in a crash.

3.4. Technique Two: Page Reorganization Indices

The B-tree modifications described above add four bytes to each key on an internal page (for a prevPtr). If keys are small, the extra four bytes will reduce B-tree fanout and increase the height of the tree. Increasing the height of the tree increases the average cost of data access.

The page reorganization algorithm reduces this loss of fanout by eliminating the prevPtr from the <*key, data*> pairs in a B-tree page. In this algorithm, however, splitting page *P* does not reclaim space on the page immediately. During the split, the DBMS copies half the keys on *P* to a new page and reorganizes *P* according to the algorithm described below (see Figure 2). After reorganization, *P*'s original keys are intact on the page. Once a sync operation successfully writes the reorganized *P* and its new peer to stable storage, the space on page *P* containing the duplicated keys is reclaimed. If the DBMS fails after *P* is written to stable storage but before *P*'s new peer is, no keys are lost. The reorganized page *P* can still be used for recovery.

The page reorganization algorithm adds the fields **prevNKeys** and **newPage** to the page header. If the prevNKeys field on a page is non-zero, the page still contains backup keys to be used in recovery. If prevNKeys is zero, the page is safe for update. Below, we describe a split of page *P* into P_a and P_b. P_a is the reorganized page. P_b is the page that will contain the new key that caused the split. Note that P_a may be either the left or the right child after the split. The newPage pointer in the reorganized page (P_a) points to P_b; newPage in P_b is nil.

A split of page *P* proceeds as follows:

(1) Two new pages are allocated. P_a is allocated in memory only; it is not backed up on the disk. P_b is allocated normally.

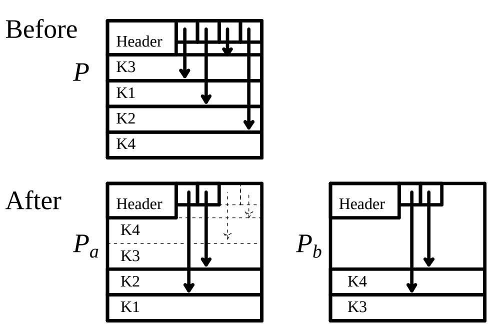

Figure 2: Page Split For Page Reorganization

(2) Half of P's keys are copied to P_a and half to P_b, just as in a normal split. The prevNKeys field on P_b is initialized to zero. On P_a, it is initialized with the number of keys on the original page P.

(3) The keys from P_b are now copied to the free space area of P_a. These keys are not *allocated* on the page, just copied into the page's free space region. A line table for the keys is set up just beyond the line table for P_a. P_a is guaranteed to have space enough for P_b's keys and line table because all of this information was stored on the original page P.

(4) The sync tokens of P_a and P_b are initialized using the global sync counter.

(5) P_a is remapped (in the in-memory buffer pool meta-data) to P's location on disk.

(6) The new key whose insertion caused the split is added to P_b. P's parent page is now updated to reflect the split.

If the next sync fails, one of five inconsistencies can occur:

(a) only P_a is written to disk (replacing P),
(b) only P_a and P_b are written (P_b is inaccessible),
(c) only the parent and P_a are written,
(d) only the parent and P_b are written,
(e) only the parent is written.

If only P_b is written, the tree is not inconsistent (but page P_b is lost).

In cases (a) and (b), the tree becomes consistent by regenerating P (assigning prevNKeys to nKeys reallocates the duplicate keys). In case (c), P_b is regenerated by copying the duplicate keys saved on P_a. In case (d), P_a is regenerated by removing the keys that are represented on P_b. In case (e), the split is repeated to generate both P_a and P_b.

Every time a key is added to or deleted from a page, the DBMS must check whether or not free space on the page can be reclaimed. If the prevNKeys field is zero, there are no extra keys stored in free space. Otherwise, the sync token on the page must be checked. There are three cases:

(1) If the sync token is the same as the global sync counter, no sync operation has occurred since the page was initialized, so the duplicate keys on the page are still required for recovery. The DBMS must block for a sync operation before the key can be added to the page.

(2) If the sync token is greater than or equal to the **last crash sync token** but different from the global sync counter, the new key can be added normally. A sync operation has definitely committed P_a and P_b, and the keys on P_a will no longer be needed for recovery.

(3) If the page sync token is less than the **last crash sync token**, we cannot immediately tell if the split was committed successfully. The DBMS has crashed since this page was written. If the page's sibling from the last split was lost in the crash, the backup keys on this page are still needed for recovery.

In the last case, the newPage pointer is used to find the sibling. If the sibling exists and has the same sync token as the current page (or a larger one), the sibling does not need to be recovered; the parent and both halves of page P made it to stable storage after the split. If the sibling is zero or has an older sync token, the sibling is out of date and must be recovered. After a new key is inserted, the prevNKeys field should be zeroed so we do not check for inconsistencies until the next page split.

3.5. Secondary Paths to Leaf Pages: B^{link}-tree

In B^{link}-tree indices, the performance of indexed scans is improved with a doubly-linked **peer pointer** chain between leaf pages with consecutive keys. The peer pointers allow scans to move from leaf page to leaf page without reading additional internal pages. Key inserts still traverse the path from root to leaf. When a page is split, the left neighbor (or right and left, in the shadow page algorithm) of the page must be re-linked so that the peer pointer path is consistent.

B^{link}-trees have more complicated failure modes than simple B-trees. There are two paths to any given leaf page; a key on the leaf page may be reached by either the peer pointer or the root-to-leaf path. Techniques like those described above could be used to correct inter-page inconsistencies in either path, but, in the worst-case failure mode, the two paths could become inconsistent with one another. For example, in Figure 3, the root-to-leaf path contains the post-split version of a given page (in bold), while the old peer pointer path contains the pre-split version of the page.

Even this worst-case failure does not actually corrupt the index unless a key is added to or deleted from one of the duplicate pages created by the failure. The transaction whose incomplete split created the duplicate paths did not commit (otherwise both paths would have been successfully written to disk). Until the first insert/delete after the failure, the duplicate pages contain the same set of valid keys.

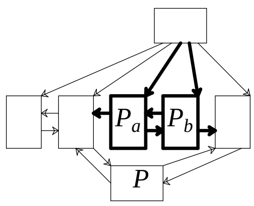

Figure 3: Worst-Case Inconsistent B^{link}-Tree

3.5.1. Detecting Inconsistencies in the Index

During a B^{link}-tree scan, the peer pointer path is checked for inter-page inconsistencies. Unfortunately, the key ranges used to detect inconsistencies in the root-to-leaf path cannot be used for the peer pointer path. On the peer pointer path, a page does not know its peer's range and cannot record it accurately unless each page is also updated when its peer splits.

To detect inconsistent peer pointer paths, we use two additional sync token fields which must be included in the page header – one associated with each peer pointer. If P1 and P2 are peer pages, P1's pointer to P2 and P2's pointer to P1 must have the same sync token associated with them. When the peer pointers are reconciled during the split, the sync tokens for the peer pointers on the neighbor pages must be reset also.

Comparing two peers' sync tokens during path traversal will detect any inconsistency in the path. If a link is broken by a crash during update, the sync tokens on adjacent pages will not agree. An inconsistent link is repaired by following the root-to-leaf path to the correct peer. If the root-to-leaf path is broken, it is repaired using one of the repair algorithms described above.

Even sync tokens do not detect the existence of two completely separate pointer paths as occurs in Figure 3. In this case, the peer pointer path is internally consistent (and the sync tokens match), but the peer pointer path is not consistent with the root-to-leaf path. Whenever a key is inserted into a page P, we must ensure that P is linked into the most recent peer pointer path.

When inserting a key into page *P*, the DBMS first checks that *P*'s split token is greater than the last crash sync token. If so, we know the page is part of a consistent peer pointer path. The path only becomes inconsistent during a system failure. Otherwise, the DBMS must follow the peer pointer path in both directions from the leaf page targeted for insert. The search stops when a page with a different sync token is discovered (page sync token *not* peer pointer sync token). If the peer pointer path is consistent until this point, the leaf page inserted into is reachable along the peer pointer path. Once this is done, we can mark the page to avoid rechecking on subsequent insertions.

3.6. Concurrency Control

The POSTGRES B-tree implementation uses a concurrency control algorithm based on Lehman-Yao [8]. In Lehman-Yao, readers and writers must descend the tree from root to leaf to find the page containing a given key. Writers ascend again as splits or deletes propagate up from the leaf. When descending, locks are not coupled; readers always release one lock before acquiring the next. When ascending, locks are coupled; the lock on a child page is released only after the correct parent page is acquired. As pointed out in [7], this algorithm is deadlock-free, since lock coupling is only used when traversing the tree in one direction.

Lehman-Yao relies on the fact that the lower-valued keys of a split page remain on the original page. Since this is not true in shadow B-trees, we add a **newPage pointer** to the B-tree page header. The newPage pointer on the original page is set to point to the new left page. Whenever a process visits a page with a non-nil newPage pointer, it traverses the link to the new page. This is analogous to the horizontal movement required in Lehman-Yao if the datum of interest was on the high half of a split page. As in B^{link}-tree peer pointer links, sync tokens are used to detect inter-page inconsistencies in the newPage pointer link. In page reorganization, we follow peer pointers as in Lehman-Yao.

We introduce a new locking protocol to ensure that peer pointers are adjusted correctly. The protocol relies on a new lock called a *split lock*. Split locks conflict only with split locks.

If a writer finds that a page must be split, it releases its write lock, acquires a split lock, and reacquires the write lock. It then splits the page. Finally, the write lock is released and peer pointers on neighboring pages are updated. The split lock is released once the peer pointers have been updated. Locking out concurrent splits guarantees that we can traverse link pointers to find neighbors and update their peer pointers. Deadlocks are impossible since processes acquire the split lock before the write lock, and acquire only one such pair in the B-tree at a time.

Concurrent access can make inter-page links temporarily inconsistent, so our algorithm must distinguish between true errors and false inconsistencies due to a concurrent update. In order to do this, we traverse a link a second time if we suspect an error. If the link is unchanged, the inconsistency is genuine and must be repaired. A temporary inconsistency between peer pointers is caused by a split of one of the two siblings. The splitter will restore consistency before releasing its write locks, so false inconsistencies are always repaired before we can traverse the link for the second time.

Finally, we must ensure that the page is no longer in use at the time it is reallocated. Suppose, for example, that a reader is descending from parent to child. It is possible for the reader to save a pointer to a child page, release the lock on the parent, and lock the child only to find that another process has split the parent and recycled the child page.

Our algorithm calls on the reader to pin the buffer containing the child page in memory before releasing the parent lock. The allocator knows not to reallocate pages in buffers with a pin count greater than one. The reader may unpin the buffer as soon as the child's lock is released. In case of page splits, a writer must keep the buffer pinned until it reascends after the update has completed. This solution does not add synchronization overhead since the buffer must be pinned in memory before use anyway. Lanin and Shasha [7] discuss two more complex techniques for solving this problem in the case of pages recycled after the last key is deleted.

4. Conventional Recovery vs. Shadowing

Either of the two techniques discussed in this paper can be used to support logical logging. In such a system, logging a record update implicitly logs any changes to related indices. If the logged change is undone or redone, any affected index records are deleted or inserted. Recovery-time insertion of a second key which points to the same record is detected and prevented. Since any unsync'd work can be lost in a failure, the B-trees must be sync'd periodically to increment sync counters and reduce the amount of work that must be redone after a failure. A conventional WAL-based storage manager, on the other hand, uses physical logging. A page split causes every key moved in the split to be logged as a delete from the original page and an insert in the new sibling page.

Logical logging has some fault tolerance advantages over conventional B-tree management. Little special case code is required for recovery. The same insert and delete operations used for normal execution are also used for recovery. Specialized recovery code includes only the code to repeat the incomplete page split after an inconsistency is detected.

More importantly, because logical logging stores a high level representation of index operations, it is harder to propagate damage caused by software errors into the log. If an internal index page is corrupted by a software error, physical logging can copy the corrupted keys into the log. During recovery, the corrupted keys will be restored to the index. Logical logging *never copies information from the index into the log*. Corruption of an index page will not be retained after a crash unless the corrupted page is saved in a checkpoint.

In [12], we compare the shadowing scheme of System R [5], the more conventional WAL-based B-trees of ARIES/IM [11], and the shadow index scheme described here. In addition to non-sequential disk layout, [11] criticizes shadowing techniques for the limits they place on concurrency. System R limited concurrency because it held transaction-duration locks on index pages. System R also allowed only one transaction to abort at a time because of potential deadlocks during undo. Neither of these are a concern in shadow indices because of the concurrency algorithms of Section 3.6.

5. Modelling The Effect of Increased Tree Heights

One performance concern regarding POSTGRES B^{link}-tree indices is that the additional space overhead they incur will increase the height of the tree, thus driving up access costs. In order to quantify this cost, we analyzed growth rates for normal, page reorganization, and shadow B^{link}-trees. The complete analysis appears in [12].

We discovered that, in practice, the space overhead for shadow index prevPtrs does not matter very much. Small trees have few levels of internal pages, so prevPtr overhead is negligible. Because of the way that the trees grow, the heights of larger normal and shadow B^{link}-trees will coincide for most index sizes. The shadow trees at these coincident heights will have less free space on internal pages, but the tree will have the same height. When index keys are large, fewer keys fit on a page and less space is lost to prevPtr overhead.

Significant differences in shadow and normal tree depth would arise only if keys were small and if the tree had many levels. However, even with the worst-case insertion order, a B^{link}-tree of either type storing four-byte keys would exceed the 2 GByte maximum size of a UNIX file before it reached five levels.

6. Performance Measurements

To measure the performance of our shadow and page reorganization index implementations, we ran two tests against each type of index. The first test built indices of three different sizes using four-byte keys. Keys were added in ascending order so as to give worst-case split performance. In each test, we measured elapsed time for each type of index. The second test retrieved 8,000 random keys from each index created in the insertion test. Keys were uniformly distributed throughout the range represented in the index. Measurements were made on a Decstation 5000/200 running Ultrix 4.0 and POSTGRES.

The times shown in Table 1 are the means of ten repetitions of each test. In all cases, the standard deviation of the measurements we took was less than 2.5% of the mean. For each time shown in the table, we show (in parentheses) the time for the test expressed on a normalized scale, where the time for the same test on the standard B^{link}-tree algorithm is defined to be one. Only time spent in the B^{link}-tree access method, and in the routines that it calls, is included in these figures. This includes time spent doing disk I/O, but does not include the cost of committing transactions. Commit cost will depend on the logging scheme chosen.

The results clearly show that the shadow algorithm is within three percent of the cost of ordinary B^{link}-trees for insertions. The higher cost is due to the added expense of verifying inter-page links in traversing the tree. For reads, the shadow tree percentages are about three and a half percent worse than ordinary B^{link}-trees.

Costs for the page reorganization algorithm are similar. Reads are between three and four percent more expensive than for the normal tree. Page reorganization

Operation	Size of Index in Keys		
B-tree Type	10,000	20,000	40,000
Inserts			
Normal	12.065 s (1.000)	24.269 s (1.000)	51.307 s (1.000)
Page Reorg	12.584 s (1.043)	25.191 s (1.038)	53.718 s (1.047)
Shadow	12.318 s (1.021)	24.924 s (1.027)	52.282 s (1.019)
8,000 Lookups			
Normal	9.122 s (1.000)	12.492 s (1.000)	19.536 s (1.000)
Page Reorg	9.441 s (1.035)	12.879 s (1.031)	20.259 s (1.037)
Shadow	9.368 s (1.027)	12.892 s (1.032)	20.200 s (1.034)

Table 1: Insert/Lookup Performance Comparison

insertions, however, are more expensive – between three and five percent higher than the cost for insertions into an ordinary B^{link}-tree. The main reason for this is that extra work must be done to order data on old pages during splits. As noted elsewhere in this paper, page reorganization is best suited to environments with low insertion rates.

The overall cost of using either index management strategy is likely to be very small for many workloads. For example, in the Wisconsin benchmark [2], POSTGRES spends only 3.6 percent of its time in the indexed access methods. Even 4.7 percent of this – our worst performance degradation – is smaller than the measurement error in the benchmark.

7. Summary

The POSTGRES DBMS relies on a no-overwrite storage system to avoid log processing during recovery. By avoiding log processing, POSTGRES recovers from failures quickly and eliminates a great deal of the complex recovery code found in most data managers. Unfortunately, concurrency requirements and inter-page pointers make the POSTGRES storage system more worthwhile for heap relations than for indices.

In this paper, we have presented two techniques for managing indices without using write-ahead log processing. Both techniques use redundant information in index pages to detect inconsistencies caused by system failures as they are encountered. Inconsistencies are removed by repeating the interrupted page split or merge operations. WAL-based data managers could also use these algorithms to avoid physical logging during page splits. Measurements and analysis of a prototype implementation suggest that the algorithms will have little impact on data manager performance.

Acknowledgements

We would like to thank Mike Stonebraker, Margo Seltzer, Wei Hong, David Bacon, Jennifer Caetta, Nat Goodman, and Ethan Munson.

References

[1] R. Bayer, C. McCreight. ‘‘Organization and Maintenance of Large Ordered Indexes,’’ *Acta Informatica*, 1(3):173-189, 1972.

[2] D. Bitton, D. DeWitt, C. Turbyfill, ‘‘Benchmarking Database Systems, a Systematic Approach,’’ *Proc. Ninth International Conf. on Very Large Databases,*, November 1983.

[3] D. Comer. ‘‘The Ubiquitous B-Tree,’’ *ACM Computing Surveys*, 11(4), 1979.

[4] R. Fagin, J. Nieverrgelt, N. Pippenger, H. Strong, ‘‘Extensible Hashing – A FastAccess Method for Dynamic Hashing,’’ *ACM Trans. on Database Systems*, 4(3):315-334, Sept. 1979.

[5] J. Gray, P. McJones, M. Blasgen, B. Lindsay, R. Lorie, T. Price, F. Putzolu, I. Traiger. ‘‘The Recovery Manager of the System R Database Manager,’’ *Computing Surveys*, 13(2):223-242, June 1981.

[6] A. Guttman. ‘‘R-Trees: A Dynamic Index Structure for Spatial Searching,’’ *Proc. ACM SIGMOD Conference*, pages 47-57, 1984.

[7] V. Lanin, D. Shasha. ‘‘A Symmetric Concurrent B-tree Algorithm,’’ *Proc. Fall Joint Computer Conference*, pages 380-389, 1986.

[8] P. Lehman, S. Yao. ‘‘Efficient Locking for Concurrent Operations on B-trees,’’ *ACM Trans. on Database Systems*, 6(4), December 1981.

[9] R. Lorie, ‘‘Physical Integrity in a Large Segmented Database,’’ *ACM Trans. on Database Systems*, 2(1):91-104, March 1977.

[10] D. Menasce, O. Landes. ‘‘Dynamic Crash Recovery of Balanced Trees,’’ *Proc. Symposium on Reliability in Distributed Software and Database Systems*, pages 131-137, July 1981.

[11] C. Mohan, F. Levine, ‘‘ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write Ahead Logging,’’ *IBM Technical Report RJ 6846*, 1989.

[12] M. Sullivan, M. Olson, ‘‘An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System,’’ *University of California, Berkeley Technical Report M91-98*, 1991.

[13] M. Stonebraker, ‘‘The POSTGRES Storage System,’’ *Proc. Very Large Data Bases Conference*, pages 289-300, September 1987.