B-trees, Shadowing, and Clones

OHAD RODEH
IBM Haifa Research Labs

B-trees are used by many file systems to represent files and directories. They provide guaranteed logarithmic time key-search, insert, and remove. File systems like WAFL and ZFS use shadowing, or copy-on-write, to implement snapshots, crash recovery, write-batching, and RAID. Serious difficulties arise when trying to use b-trees and shadowing in a single system.

This article is about a set of b-tree algorithms that respects shadowing, achieves good concurrency, and implements cloning (writeable snapshots). Our cloning algorithm is efficient and allows the creation of a large number of clones.

We believe that using our b-trees would allow shadowing file systems to better scale their on-disk data structures.

1. INTRODUCTION

B-trees [Bayer and McCreight 1972] are used by several file systems [Sweeny et al. 1996; Best 2002; Menon et al. 2003; Reiser 2004] to represent files and directories. Compared to traditional indirect blocks [McKusick et al. 1984], b-trees offer guaranteed logarithmic-time key-search, insert, and remove. Furthermore, b-trees can well represent sparse files.

Shadowing is a technique used by file systems like WAFL [Hitz et al. 1994] and ZFS [Henson et al. 2003] to ensure atomic update to persistent data-structures. It is a powerful mechanism that has been used to implement snapshots, crash recovery, write-batching, and RAID. The basic scheme is to look at the file system as a large tree made up of fixed-sized pages. Shadowing means
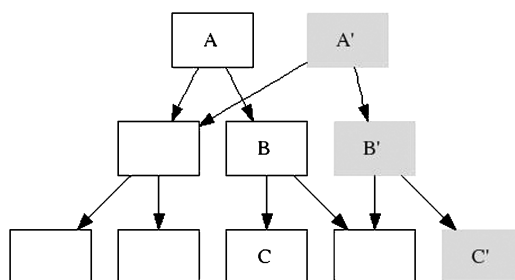
Fig. 1. Modifying a leaf requires shadowing up to the root.

that to update an on-disk page, the entire page is read into memory, modified, and later written to disk at an alternate location. When a page is shadowed, its location on disk changes, and this creates a need to update (and shadow) the immediate ancestor of the page with the new address. Shadowing propagates up to the file system root. We call this kind of shadowing *strict* to distinguish it from other forms of shadowing [Gray and Reuter 1993]. Figure 1 shows an initial file system with root A that contains seven nodes. After leaf node C is modified, a complete path to the root is shadowed, creating a new tree rooted at A'. Nodes A, B, and C become unreachable and will later be deallocated.

In order to support snapshots, the file system allows having more than a single root. Each root node points to a tree that represents a valid image of the file system. For example, if we were to decide to perform a snapshot prior to modifying C, then A would have been preserved as the root of the snapshot. Only upon deletion of the snapshot would it be deallocated. The upshot of this is that pages can be shared between many snapshots; indeed, whole subtrees can be shared between snapshots.

The b-tree variant typically used by file systems (e.g., XFS [Sweeny et al. 1996] and JFS [Best 2002]) is b+-trees. In a b+-tree, leaf nodes contain data values and internal nodes contain only indexing keys. Files are typically represented by b-trees that hold disk extents in their leaves. Directories are represented by b-trees that contain variable-sized directory entries in their leaves. In this work we focus on these types of b-tree usage.

B-trees used in file systems have to be persistent and recoverable. The methods described in the literature [Mohan and Levine 1992; Lomet and Salzberg 1992] use a bottom-up update procedure. Modifications are applied to leaf nodes. Rarely, leaf nodes split or merge, in which case changes propagate to the next level up. This can occur recursively and changes can propagate high up into the tree. Leaves are chained together to facilitate rebalancing operations and range lookups. There are good concurrency schemes allowing multiple threads to update a tree; the best is currently b-link trees [Lehman and Yao 1981].

The main issues when trying to apply shadowing to the classic b-tree are as follows.

—*Leaf Chaining*. In a regular b-tree, leaves are chained together. This is used for tree rebalancing and range lookups. In a b-tree that is updated using
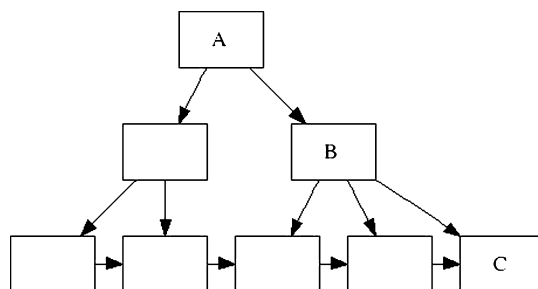
Fig. 2.   A tree whose leaves are chained together. The rightmost leaf is C and B is its immediate ancestor. If C is modified, the entire tree has to be shadowed.
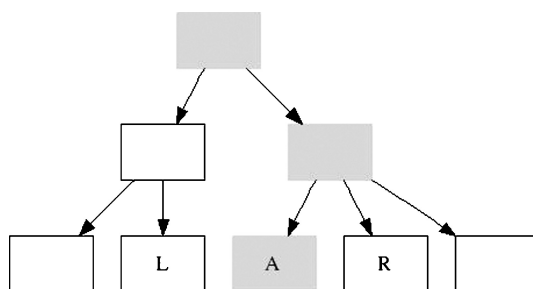

Fig. 3.   Removing a key and effects of rebalancing and shadowing.

copy-on-write, leaves cannot be linked together. For example, Figure 2 shows a tree whose rightmost leaf node is C and where the leaves are linked from left to right. If C is updated, the entire tree needs to be shadowed. Without leaf pointers, only C, B, and A require shadowing.

Without links between leaves, much of the b-tree literature becomes inapplicable.

—Concurrency.　In a regular b-tree, in order to add a key to a leaf *L*, in most cases, only *L* needs to be locked and updated. When using shadowing, every change propagates up to the root. This requires exclusive locking of top nodes, making them contention points. Shadowing also excludes b-link trees because b-link trees rely on in-place modification of nodes as a means to delay split operations.

—Modifying a Single Path.　Regular b-trees shuffle keys between neighboring leaf nodes for rebalancing purposes after a remove-key operation. When using copy-on-write this habit could be expensive. For example, in Figure 3a tree with leaf node A and neighboring nodes R and L is shown. A key is removed from node A and the modified path includes 3 nodes (shadowed in the figure). If keys from node L were moved into A, then an additional tree-path would need to be shadowed. It is better to shuffle keys from R.

This work describes the first b-tree construction that can coexist with shadowing while providing good concurrency. This is an important result because b-trees and shadowing are basic file system techniques.

Our cloning algorithm improves upon the state-of-the-art. We support a large number of clones and allow good concurrency when accessing multiple clones that share blocks. Cloning is a fundamental user requirement; supporting it efficiently is important in today's file systems.

It is problematic to compare this aritcle to previous work because there is not a lot of literature on the two relevant file systems, ZFS and WAFL. There is only one short paper on ZFS [Henson et al. 2003] and the paper on WAFL [Hitz et al. 1994] was written more than 12 years ago. Clearly, many changes have been made to WAFL since then. The author wishes to apologize in advance for any improvements made to these systems of which he was not aware at the time this article was written.

The gist of our approach is: (1) to use top-down b-trees instead of bottom-up b-trees and thus integrate well with a shadowing system; (2) remove leaf chaining; and (3) use lazy reference-counting for the free-space map and thus support many clones.

The rest of this article is organized as follows: Section 2 is about related work, Section 3 discusses recoverability, Section 4 describes the basic algorithms, Section 5 describes cloning, Section 6 describes the runtime system, Section 7 shows performance, Section 8 discusses future work, and Section 9 summarizes.

2. RELATED WORK

There is a vast body of literature on b-trees [Comer 1979], their concurrency schemes, and update methods. We cannot hope to describe all of the related work, so we limit ourselves to several important cases. The difficulty that this work grapples with is the additional constraint of strict shadowing. To the best of the author's knowledge, this is the first work that deals with strict shadowing in conjunction with concurrency, recoverability, and cloning.

There is an alternate style of copy-on-write used in some databases [Rosenberg et al. 1990; Gray and Reuter 1993]. The main idea is to give database pages virtual addresses that never change. There is a mapping table that maps virtual addresses to physical locations on disk. In order to modify a page, it is copied to an alternate location on disk and the mapping table is modified to point to its new location. This shadowing method does not suffer from the limitations inherent to a strict-shadowing system. When a b-tree leaf is modified, its ancestors do not need to be shadowed. Furthermore, leaf chaining is allowed because internode pointers use virtual addresses. A scheme with good concurrency such as a b-link [Lehman and Yao 1981] tree can be used. The downside is that the additional mapping table can be quite large, and efficient access to it becomes critical for performance. Furthermore, additional effort is required to implement snapshots.

Some databases and file systems use b-trees that are updated in place. An important method for implementing persistent recoverable b-trees is ARIES [Mohan and Levine 1992]. The main idea is to use a bottom-up b-tree update procedure. Updates are made to leaves, and in case of overflow/underflow changes, propagate up the tree. The vast majority of updates affect a single leaf.

In terms of locking, the minimal affected subtree is locked for the duration of the operation. This provides good concurrency.

This work makes use of top-down b-trees; these were first described in Guibas and Sedgewick [1978], Mond and Raz [1985], and Lanin and Shasha [1986].

It is possible to use disk extents instead of fixed-sized blocks [Sweeny et al. 1996; Best 2002; Reiser 2004]. We have experimented with using extents. The resulting algorithms are similar in flavor to those reported here and we do not describe them for brevity.

B-trees also support range operations. For example, range lookup is useful when reading file-ranges or contiguous directory entries. Remove-range is useful for truncating a file or removing a range in the middle of it. We do not report on these operations for brevity; there is a technical report describing our algorithms [Rodeh 2006].

In our solution we abandon leaf chaining. This design choice can disadvantage range lookups and rebalancing opportunities. Graefe [2004] describes how leaf chaining does not really help range lookups. This is because in order to effectively perform a sequential scan of the b-tree, multiple leaf nodes have to be prefetched from disk in parallel. This can be achieved by traversing the tree from the top and issuing prefetch requests for several leaf nodes concurrently. Leaf chaining is of no help for this method of operation. Rebalancing opportunities are not much affected by the lack of leaf chaining because the b-tree fanout is normally very large. Therefore, the fraction of nodes that are missing a left or right sibling is very small.

Snapshots have a variety of uses: recovering accidentally deleted files, reverting back to known good state of the file system after corruption, data mining, backup, and more. Clones, or writeable snapshots, are an extension of the snapshot concept where the snapshot can also be overwritten with new data. A clone can be used to: (1) create a point in time copy of an existing volume, and (2) try out experimental software on it.

Snapshots are supported by many file systems. Clones are a more recent user requirement and are not widely supported as of yet. The details of how many snapshots/clones can be created and how efficiently they are supported varies widely between implementations.

Continuous data protection (CDP) is a general name for storage systems that have the ability to keep the entire history of a volume. CDP systems support two main operations: (1) going back in time to any point in history in read-only mode; and (2) reverting back to a previous point in time and continuing the update from there.

CDP systems differ in granularity of the history they keep. Some systems are able to provide a granularity of every I/O, others per second, still others per hour.

Supporting a large number of clones is a method of implementing coarse granularity CDP. Clearly, it is infeasible to create a snapshot after each file system operation. However, a snapshot every five minutes is perhaps reasonable. We were interested in trying to support a large number of clones, thereby supporting snapshots, clones, and coarse-granularity CDP with a single tool.

There are other approaches to retaining file system histories. For example, the comprehensive versioning file system (CVFS) [Soules et al. 2003] attempts to efficiently retain earlier versions of files and directories. Several techniques are used to improve efficiency for the file system metadata. Multiversion b-trees are used to store directories. Indirect blocks and i-nodes are encoded using a journal. This means that viewing an older version of a file requires replaying a log in reverse and undoing changes made to the file. The CVFS file system is log structured. Pages have virtual addresses, allowing moving pages around by the file system cleaner without breaking interpage pointers.

The major differences with respect to our work are the following.

—CVFS does not adhere to strict shadowing; rather, it uses the virtual address method. In this way CVFS sidesteps the problems with which we were faced.
—CVFS retains a very good history of changes made to the file system; however, access to the history is read-only. Clones are not supported.

The WAFL system [Hitz et al. 1994] has a cloning algorithm that is closest to ours. Although the basic WAFL paper discusses read-only snapshots, the same ideas can be used to create clones. WAFL has three main limitations which we improve upon, described next.

(1) WAFL uses a free-space map where a bit is used to represent the fact that a block belongs to a particular snapshot. This means that in order to support 256 snapshots, 32 bytes are required in the free-space map per block.
(2) In WAFL, in order to create or delete a snapshot, a pass on the entire free-space is required.
(3) The granularity of a clone operation is an entire volume.

By contrast, in our algorithm the following holds.

(1) A single byte per block is sufficient for supporting 256 clones.
(2) Only the children of the root of a b-tree are involved in snapshot creation. Free-space map operations are performed gradually through the lifetime of the clone.
(3) A single b-tree can be cloned. This allows cloning a volume, but also cloning a single file.

There is a long-standing debate as to whether it is better to shadow or to write-in-place. The wider discussion is beyond the scope of this article. This work is about a b-tree technique that works well with shadowing.

3. RECOVERABILITY

Shadowing file systems [Hitz et al. 1994; Henson et al. 2003] ensure recoverability by taking periodic checkpoints, and logging commands in between. A checkpoint includes the entire file system tree; once a checkpoint is successfully written to disk, the previous one can be deleted. If a crash occurs, the file system goes back to the last complete checkpoint and replays the log.

(a) initial file system tree

(b) modifications
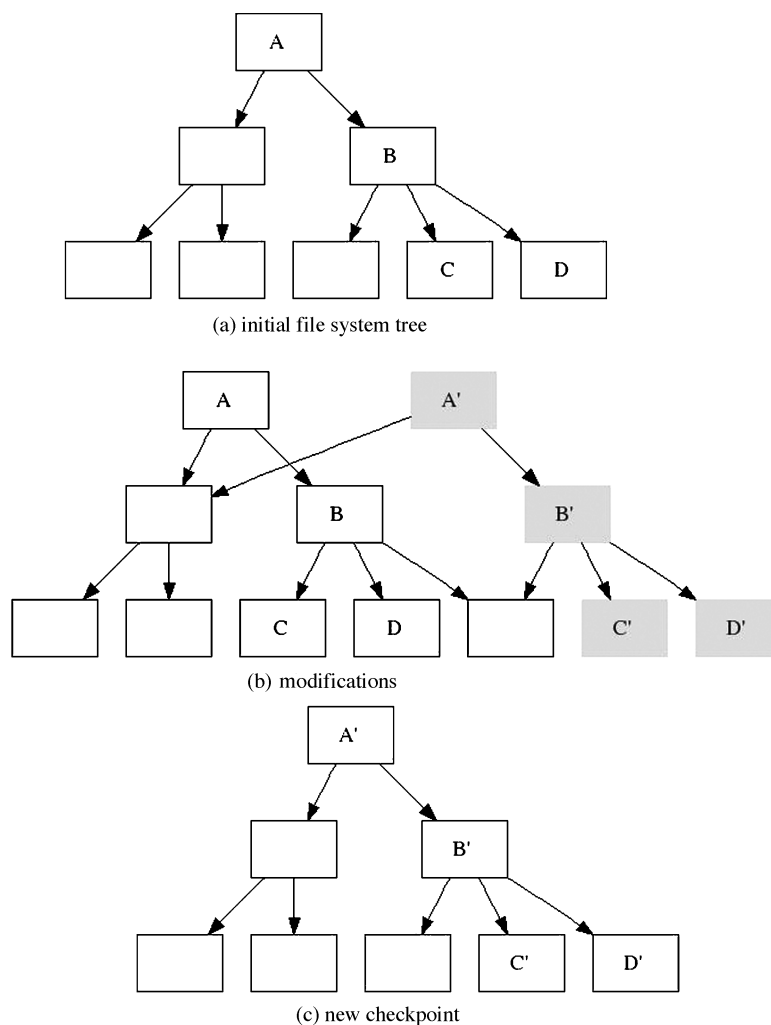
(c) new checkpoint

Fig. 4.   Checkpoints.

For example, Figure 4(a) shows an initial tree. Figure 4(b) shows a set of modifications marked in gray. Figure 4(c) shows the situation after the checkpoint has been committed and unreferenced pages have been deallocated.

The process of writing a checkpoint is efficient because modifications can be batched and written sequentially to disk. If the system crashes while writing a checkpoint, no harm is done; the previous checkpoint remains intact.

Command logging is attractive because it combines into a single log-entry a set of possibly complex modifications to the file system.

This combination of checkpointing and logging allows an important optimization for the shadow-page primitive. When a page belonging to a checkpoint is first shadowed, a cached copy of it is created and held in memory. All modifications to the page can be performed on the cached shadow copy. Assuming there

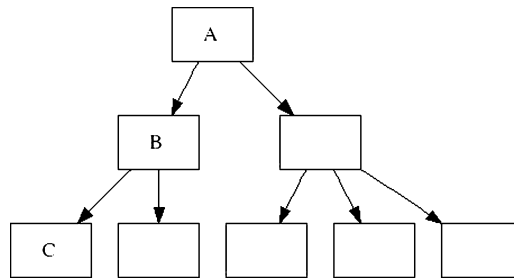Fig. 5. Three types of nodes in a tree; A is a root node, B is an index node, and C is a leaf node. Leaves are not linked together.
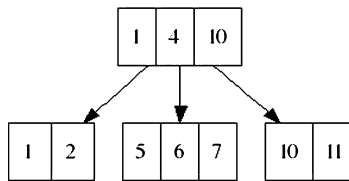

Fig. 6. A b-tree with two levels.

is enough memory, the dirty page can be held until the next checkpoint. Even if the page needs to be swapped out, it can be written to the shadow location and then paged to/from disk. This means that additional shadows need not be created.

Checkpoints are typically implemented using the clone primitive. They are simply clones that users cannot access. Our implementation of clones is described in Section 5.

4. BASE ALGORITHMS

The variant of b-trees that is used here is known as *b+-trees*. In a b+-tree leaf nodes contain key-data pairs, and index nodes contain mappings between keys and child nodes; see Figure 5. The tree is composed of individual nodes where a node takes up 4KB of disk space. The internal structure of a node is based on Lomet [2001]. There are no links between leaves.

Shadowing a page can cause the allocation of a page on disk. When a leaf is modified, all the nodes on the path to the root need to be shadowed. If trees are unbalanced then the depth can vary depending on the leaf. One leaf might cause the modification of 10 nodes, another only 2. Here, all tree operations maintain a perfectly balanced tree; the distance from all leaves to the root is the same.

The b-trees use a minimum-key rule. If node N_1 has a child node N_2, then the key in N_1 pointing to N_2 is smaller or equal to the minimum of (N_2). For example, Figure 6 shows an example where integers are the keys. In this diagram and throughout this document, data values that should appear at the leaf nodes are omitted for simplicity.

B-trees are normally described as having between b and $2b - 1$ entries per node. Here, these constraints are relaxed and nodes may contain between b and

3	7

Fig. 7.   A b-tree containing only a root.

$2b + 1$ entries, where $b \geq 2$. For performance reasons it is desirable to increase the upper bound to $3b$; however, in this section we limit ourselves to the range $[b \ldots 2b + 1]$.

A proactive approach to rebalancing is used. When a node with $2b+1$ entries is encountered during an insert-key operation, it is split. When a node with b entries is found during a remove-key operation, it is *fixed*. Fixing means either moving keys into it or merging it with a neighbor node so it will have more than b keys. Proactive fix/split simplifies tree-modification algorithms as well as locking protocols because it prevents modifications from propagating up the tree. However, care should be taken to avoid excessive split/fix activity. If the tree constraints were b and $2b - 1$, then a node with $2b - 1$ entries could never be split into two legal nodes. Furthermore, even if the constraints were b and $2b$, a node with $2b$ entries would split into two nodes of size b which would immediately need to be merged back together. Therefore, the constrains are set further away, enlarging the legal set of values. In all the examples in this section $b = 2$ and the set of legal values is $[2 \ldots 5]$.

During the descent through the tree, *lock-coupling* [Bayer and Schkolnick 1977] is used. Lock-coupling (or *crabbing*) means locking children before unlocking the parent. This ensures the validity of a tree path that a task is traversing without prelocking the entire path. Crabbing is deadlock free.

When performing modifying operations such as insert-/remove-key, each node on the path to the leaf is shadowed during the descent through the tree. This combines locking, preparatory operations, and shadowing into one downward traversal.

4.1 Create

In order to create a new b-tree, a root page is allocated and formatted. The root page is special; it can contain zero to $2b + 1$ entries. All other nodes have to contain at least b entries. Figure 7 presents a tree that contains a root node with 2 entries.

4.2 Delete

In order to erase a tree, it is traversed and all the nodes and data are deallocated. A recursive postorder traversal is used.

An example for the postorder delete pass is shown in Figure 8. A tree with eight nodes is deleted.

4.3 Insert-Key

Insert-key is implemented with a proactive split policy. On the way down to a leaf, each full index node is split. This ensures that inserting into a leaf will, at most, split the leaf. During the descent lock-coupling is used. Locks are taken in exclusive mode. This ensures the validity of a tree path that a task is traversing.
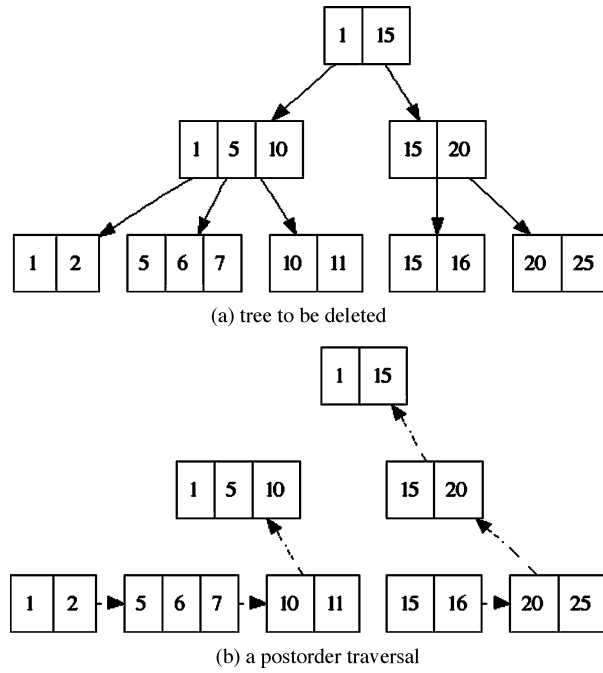
(a) tree to be deleted

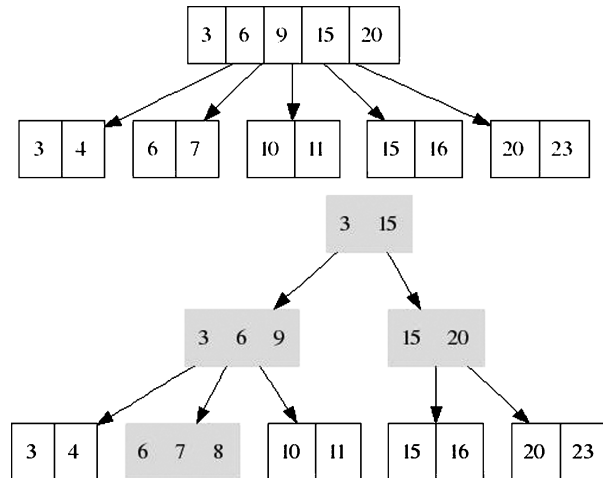(b) a postorder traversal

Fig. 8.   Deleting a tree.

Fig. 9.   Inserting key 8 into a tree. Gray nodes have been shadowed. The root node has been split and a level was added to the tree.

Figure 9 shows an example where key 8 is added to a tree. Node [3, 6, 9, 15, 20] is split into [3, 6, 9] and [15, 20] on the way down to leaf [6, 7]. Gray nodes have been shadowed.
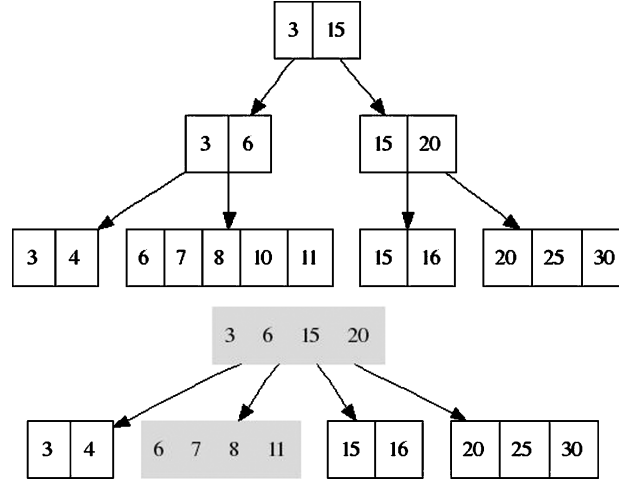
Fig. 10.   Removing key 10 from a tree. Gray nodes have been shadowed. The two children of the root were merged and the root node was replaced.

4.4 Lookup-Key

Lookup for a key is performed by an iterative descent through the tree using lock-coupling. Locks are taken in shared mode.

4.5 Remove-Key

Remove-key is implemented with a proactive merge policy. On the way down to a leaf, each node with a minimal amount of keys is fixed, ensuring it will have at least $b + 1$ keys. This guarantees that removing a key from the leaf will, at worst, affect its immediate ancestor. During the descent in the tree lock-coupling is used. Locks are taken in exclusive mode.

For example, Figure 10 shows a remove-key operation that fixes index-node [3, 6] by merging it with its sibling [15, 20].

4.6 Resource Analysis

This section analyzes the requirements of each operation in terms of memory and of disk pages.

For insert-/remove-key, three memory pages are needed in the worst case; this happens if a node needs to be split or fixed during the downward traversal. The number of modified disk pages can be $2 \times$ tree-depth. Since the tree is balanced, the tree depth is in the worst case equal to $log_b(N)$, where N is the number of keys in the tree. In practice, a tree of depth 6 is more than enough to cover huge objects and object catalogs.

For lookup-key, two memory pages are needed. For lookup-range, three memory pages are needed.

4.7 Comparison to Standard B-Trees

A top-down b-tree with bounds $[b, 2b+1]$ is, on average, no worse than a bottom-up b-tree with bounds $[b + 1, 2b]$. The intuition is that a more aggressive top-down algorithm would never allow nodes with b or $2b + 1$ entries; such nodes would be immediately split or fixed. This means that each node would contain between $b + 1$ and $2b$ entries. This is more or less equivalent to a bottom-up b-tree with $b' = b + 1$.

In practice, the bounds on the number of entries in a node are expanded to $[b, 3b]$. This improves performance because it means that there are few spurious cases of split/merge. The average capacity of a node is around $2b$.

5. CLONES

This section builds upon Section 4 and adds the necessary modifications so that clones will work.

There are several desirable properties in a cloning algorithm. Assume T_p is a b-tree and T_q is a clone of T_p, then the following should hold.

—*Space Efficiency*. T_p and T_q should, as much as possible, share common pages.
—*Speed*. Creating T_q from T_p should take little time and overhead.
—*Number of Clones*. It should be possible to clone T_p many times.
—*Clones as First-Class Citizens*. It should be possible to clone T_q.

A trivial algorithm for cloning a tree is copying it wholesale. However, this neither provides space efficiency nor speed. The method proposed here does not copy the entire tree and has the desired properties.

The main idea is to use a free-space map that maintains a reference count (*ref-count*) per block. The ref-count records how many times a page is pointed to. A zero ref-count means that a block is free. Essentially, instead of copying a tree, the ref-counts of all its nodes are incremented by one. This means that all nodes belong to two trees instead of one; they are all shared. However, instead of making a pass on the entire tree and incrementing the counters during the clone operation, this is done in a lazy fashion.

Throughout the examples in this section, trees T_p and T_q are used. Tree T_p has root P and tree T_q has root node Q. Nodes whose ref-count has changed are marked with diagonals, and modified nodes are colored in light gray. In order to better visualize the algorithms, reference counters are drawn inside nodes. This can be misleading, since the ref-counters are physically located in free-space maps.

5.1 Create

The algorithm for cloning a tree T_p consists of the following.

(1) Copy the root node of T_p into a new root.
(2) Increment the free-space counters for each of the children of the root by one.

An example for cloning is shown in Figure 11. Tree T_p contains 7 nodes and T_q is created as a clone of T_p by copying the root P to Q. Both roots point to the

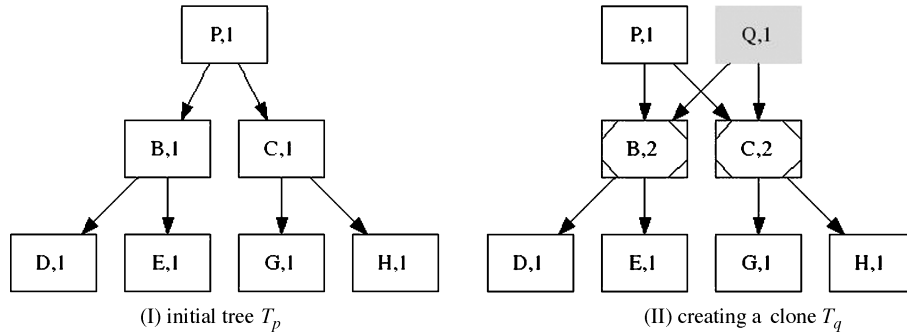(I) initial tree T_p (II) creating a clone T_q

Fig. 11. Cloning a b-tree.

shared children: B and C. The reference counters for B and C are incremented to 2.

Notice that in Figure 11(II) nodes D, E, G, and H have a ref-count of one, although they belong to two trees. This is an example of lazy reference-counting.

5.2 Lookup

The lookup-key and lookup-range algorithms are unaffected by the modification to free-space maps.

5.3 Insert-Key and Remove-Key

Changing the way the free-space works impacts the insert-key and remove-key algorithms. It turns out that a subtle change is sufficient to get them to work well with free-space ref-counts.

Before modifying a page, it is "marked-dirty". This lets the runtime system know that the page is about to be modified and gives it a chance to shadow the page if necessary.

The following procedure is followed when marking-dirty a clean page N.

(1) If the reference count is 1 nothing special is needed. This is no different than without cloning.
(2) If the ref-count is greater than 1 and page N is relocated from address L_1 to address L_2, the ref-count for L_1 is decremented and the ref-count for L_2 is made 1. The ref-count of N's children is incremented by 1.

For example, Figure 12 shows an example of two trees, T_p and T_q, that start out sharing all their nodes except for the root. Initially, all nodes are clean. A key is inserted into tree q leaf H. This means that a downward traversal is performed and nodes Q, C, and H are shadowed. In stage (II) node Q is shadowed. Its ref-count is 1, so nothing special is needed. In stage (III) node C is shadowed; this splits C into two versions: one belonging to T_p and the other to T_q, each with a ref-count of 1. The children of C are nodes H and G, and their ref-count is incremented to 2. In stage (IV) node H is shadowed; this splits H into two separate versions, each with ref-count 1.
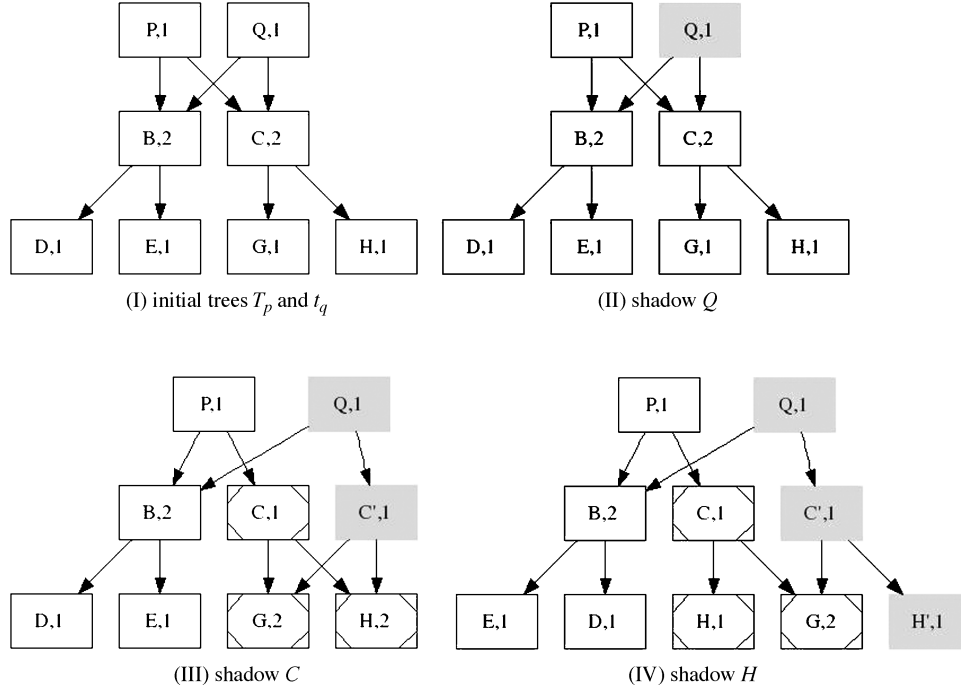
Fig. 12.   Inserting into a leaf node breaks sharing across the entire path.

Performing the mark-dirty primitive in this fashion allows delaying the ref-count operations. For example, in Figure 12(I), node C starts out with a ref-count of 2. At the end of the insert operation there are two versions of C, each with a ref-count of 1. Node G starts out with a ref-count of 1 because it is shared indirectly between T_p and T_q. At the end of the operation, it has a ref-count of 2 because it is pointed-to directly from nodes in T_p and T_q.

This modification to the mark-dirty primitive gets the insert-key and remove-key algorithms to work.

5.4 Delete

The delete algorithm is also affected by the free-space ref-counts. Without cloning, a postorder traversal is made on the tree and all nodes are deallocated. In order to take ref-counts into account, a modification has to be made. Assume tree T_p is being deleted and that during the downward part of the postorder traversal, node N is reached. Then we proceed as follows.

(1) If the ref-count of N is greater than 1, then decrement the ref-count and stop downward traversal. The node is shared with other trees.
(2) If the ref-count of N is 1, then it belongs only to T_p. Continue downward traversal and on the way back up, deallocate N.

Figure 13 shows an example where T_q and T_p are two trees that share some of their nodes. Tree T_q is deleted. This frees nodes Q, X, and Z and reduces the ref-count on nodes C and Y to 1.
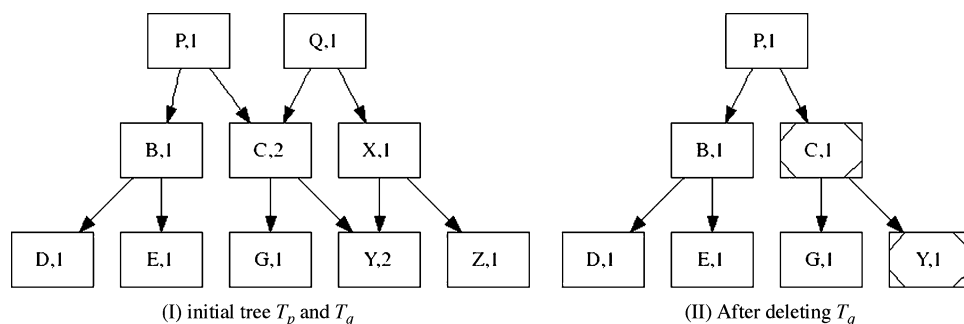
Fig. 13. Deleting a b-tree rooted at Q.

The caption shows two tree diagrams: (I) initial tree T_p and T_q, and (II) After deleting T_q.

5.5 Resource and Performance Analysis

The modifications made to the basic algorithms do not add b-tree node accesses. This means that the worst-case estimate on the number of memory pages and number of disk blocks used per operation remains unchanged. The number of free-space accesses increases. This has the potential to significantly impact performance.

Two observations make this unlikely, as detailed next.

—Once sharing is broken for a page and belongs to a single tree, there are no additional ref-count costs associated with it.

—The vast majority of b-tree pages are leaves. Leaves have no children and therefore do not incur additional overhead.

The test framework used in this work includes a free-space map that resides in memory. This does not allow a serious attempt to investigate the costs of a large free-space map. Furthermore, even a relatively large b-tree that takes up a gigabyte of disk space can be represented by a 1MB free-space map that can be held in memory. Therefore, investigating this issue remains for future work.

Concurrency remains unaffected by ref-counts. Sharing on any node that requires modification is quickly broken and each clone gets its own version. There is no contention on the free-space counters because each thread briefly touches a counter and then releases it.

5.6 Comparison to WAFL Free-Space

It is interesting to compare lazy free-space counters with the WAFL free-space implementation. This is a somewhat unfair comparison because the WAFL [Hitz et al. 1994] free-space algorithm dates back to 1994. Undoubtedly, many improvements have been made to it since. Unfortunately, none of this work has been published.

We examine four use-cases: cloning a volume, deleting a volume, normal updates (no clone operations), and cloning a single file. We also compare the space taken up by the free-space map.

When a volume is cloned, WAFL makes a pass on the entire free-space and sets bits. The ref-count algorithm increases the ref-counts only for children of the root.
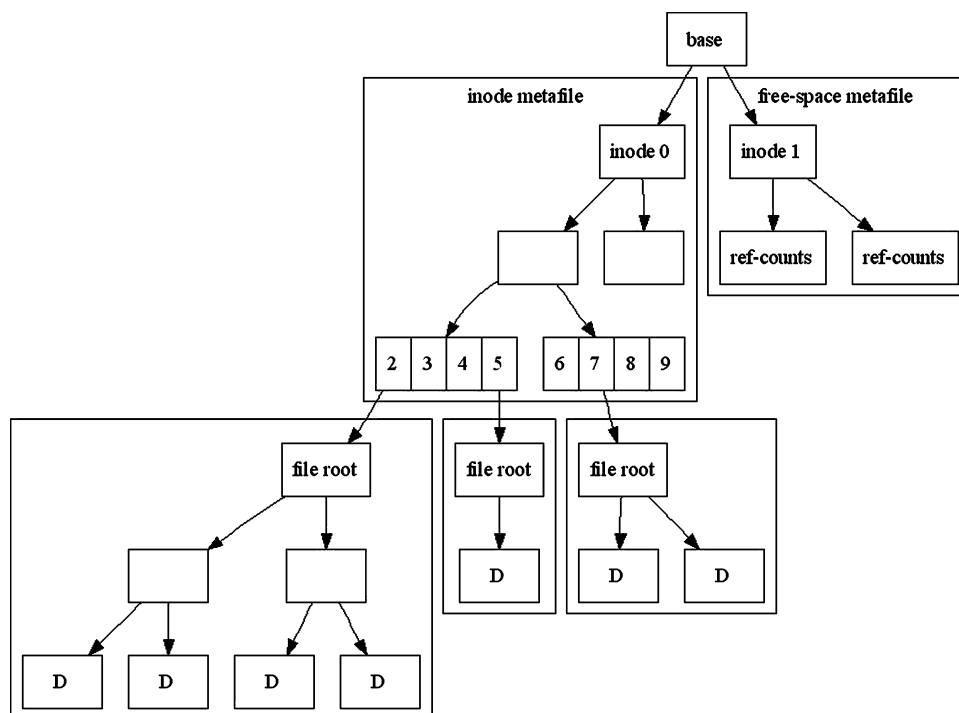
Fig. 14.   The basic structure of a β file system. A metafile holds the i-nodes. An additional metafile holds the free-space ref-counts. Each i-node points to file data, if it exists. I-nodes 2, 5, and 7 have data. The leaf nodes of files contain data and are marked with a D, and all other nodes contain internal file system metadata.

When a volume p is deleted, WAFL makes a pass on the entire free-space and sets the volume bit to zero. The ref-count algorithm traverses only those nodes that belong only to p and decrements ref-counters.

During normal runtime, WAFL incurs no additional costs due to free-space maps. The ref-count algorithm incurs additional costs when updates occur.

WAFL uses 32 bytes per block to represent 256 clones. The ref-count algorithm needs only 1 byte per block to represent the same number of clones.

In order to clone a single file, WAFL has to clone an entire volume. With reference counters an individual file can be cloned by cloning its root.

5.7 A Ref-Count Based File System

To allow a more in-depth view of how b-trees and ref-counts can be applied to file system technology, we examine a hypothetical file system which we shall call β_{FS}. β_{FS} is a WAFL system that has been modified to use b-trees instead of traditional indirect blocks, and ref-counts instead of its free-space bitmaps. Figure 14 shows the basic structure of the β file system. The root i-node (i-node zero) points to a metafile that holds the i-nodes of the volume; each i-node points to the root of a b-tree holding file data. An additional metafile (i-node one) holds the free-space counters.
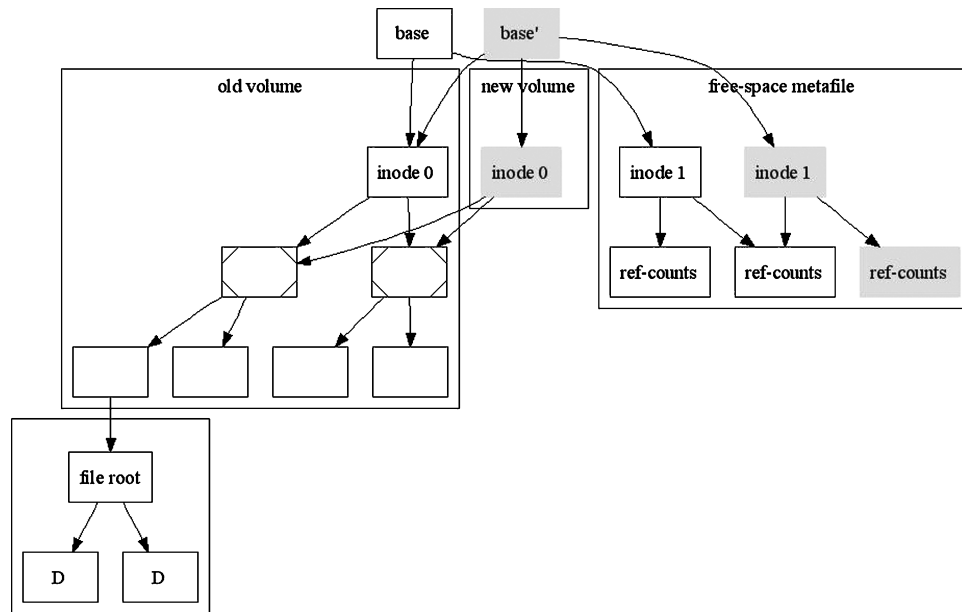
Fig. 15.   Cloning a volume. A new i-node metaile is created by cloning the original metafile. The ref-counts for the children are incremented and changes are made to the free space. Nodes whose ref-count have been changed are marked with diagonals. Changes ripple up to the top of the file system.

In order to clone a volume, the b-tree clone operation is performed on its root i-node; see Figure 15. In order to clone a file, the b-tree clone operation is performed on the b-tree of the file and a new i-node is added; see Figure 16.

The free space is stored in a metafile common to all volumes. The data of the metafile consists of arrays of reference counters: a counter per each disk block. In order to modify a ref-count, the path to it is shadowed. Figure 17 shows an example of a free-space map for blocks 0 through 11; the counter for block 10 is modified.

5.8 Maintaining Consistency for the Free-Space Map

The free-space map is a critical data structure; its reference counters are accessed frequently when shadowing pages. It is important to ensure the following.

—Accessing and updating ref-counts can be done by multiple threads with good concurrency.
—While updating ref-counts, the file system consistency remains intact.

In order to modify a ref-count, the file system needs to shadow the path to the block containing it. Locks on free-space pages are held only briefly, for the duration of the in-memory modification. They are not held during I/O. This ensures low contention over the ref-counts.
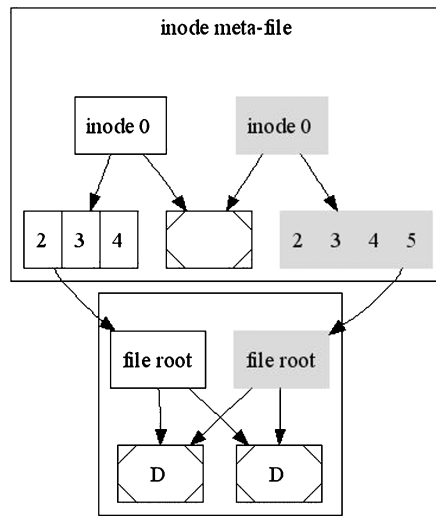
Fig. 16. Cloning a file. I-node 2 points to the original file. The clone operation includes creating a new i-node with the new file name (i-node 5), and cloning the b-tree holding the data.
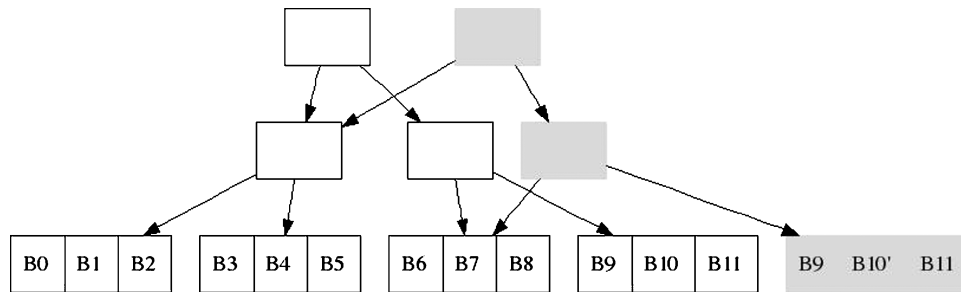
Fig. 17. Updating the ref-count for block 10. The indirect blocks and the data block containing B10 are shadowed. The value of B10 is modified to B10'.

The free-space map is updated using shadowing. This means that using checkpoints and logical logging ensures its consistency. For example, assume that the previous checkpoint is P and that various changes have been made to the file system free-space and data structures since P. A crash occurs and recovery is initiated. The recovery sequence consists of going back to P and replaying all log records.

5.9 Late Allocation and Fragmentation

Shadowing file systems typically employ *late allocation* in order to improve free-space performance. The idea is to delay allocation until the last possible moment, which is when dirty blocks can no longer be cached in memory but must be flushed to disk. In practice, allocation can be delayed until a checkpoint occurs. Late binding allows allocating multiple dirty blocks on a long disk extent and writing them out to disk in one I/O; this is also called *write-batching*. Write-batching can significantly improve the time it takes to perform a checkpoint.

However, batching requires the existence of contiguous extents of free space. In other words, the free space must remain unfragmented.

As the file system ages it naturally starts to fragment. Clones complicate defragmentation because there may be multiple pointers to a block; when the block is moved the pointers have to be updated with the new address. How do we locate the pointers, since they can be on any block on disk? A possible solution is to employ virtual addresses for pages and then move pages around when needed. This technique is used by CVFS [Soules et al. 2003].

6. THE RUNTIME SYSTEM

A minimal runtime system was created for the b-tree. The rational, is to focus on the tree algorithms themselves, rather than any fancy footwork that can be performed by a shadowing file system.

The b-tree is split into 4KB pages that are paged to/from disk. A page cache is situated between the b-tree and the disk; it can cache both clean and dirty pages. A simple clock scheme is implemented, and no attempt is made to coalesce pages written to disk into large writes, no prefetching is performed. In order to shadow a page P, the page is first read from disk and put into the cache. As long as P stays in cache, it can be modified in memory. Once there is memory pressure, P is written to disk. If P belongs to more than one clone, it has to be written to an alternate location; otherwise, it can be written in place. This way, the cache absorbs much of the overhead of shadowing, especially for heavily modified pages.

The free space was implemented with a simple in-memory map split into areas that can be separately locked. There is a ref-count per block. Investigating a free-space map that is paged to disk remains for future work.

It is assumed that the first step in processing a file system operation is to write a logical log record describing the operation to NVRAM. This means that b-tree operations are recoverable through log replay. We did not have access to an NVRAM card and so we did not emulate that part of the system. Writing to NVRAM is assumed to be relatively fast and inexpensive on the order of 10 microseconds. Therefore, we believe that the addition of a log would not significantly impact our performance results.

The only use of NVRAM in the system is for logging; it is not required for any other reason. Note that if NVRAM is not used, the resulting system moves from one checkpoint to another; if an error occurs between checkpoints, all changes in between are lost.

The implementation platform was a Linux 2.6 operating system with a standard pthread threading package. Our code was implemented in user space, and did not contain any in-kernel parts. The processors used were Intel Xeons.

7. PERFORMANCE

B-trees are a fundamental part of the file system, and appear in directories and files, and must perform well under a wide range of workloads. We attempted to choose a representative set of workloads and parameters in our empirical evaluation.

Table I. Basic Disk-Subsystem Capabilities

#threads	op.	time per op.(ms)	ops per second
10	read	N/A	1217
	write	N/A	640
	R + W	N/A	408
1	read	3.8	256
	write	16.8	59
	R + W	16.9	59

Three workloads were used: (1) read a random page, (2) write a random page, and (3) read and write a random page. Using 10 threads increases the number of operations per second by a factor of 5 to 6.

In Srinivasan and Carey [1993] there is a prediction that top-down algorithms will not work well. This is because every tree modification has to exclusively lock the root and one of its children, creating a serialization point. We found this not to be a problem in practice. What happens is that the root and all of its children are almost always cached in memory; therefore, the time it takes to pass the root and its immediate children is very small. Operations spend most of the time waiting for I/O on the leaves.

In the experiments reported in this section the entries are of size 16 bytes: 8 bytes for a key and 8 bytes for data. A 4KB node can contain up to 235 such entries. Choosing other key and data sizes yielded similar performance results.

The testbed used in the experiments was a single machine connected to a DS4500 disk controller through Fiber-Channel. The machine was a dual-CPU Xeon (Pentium4) 2.4Ghz with 2GB of memory. It ran a Linux-2.6.9 operating system. The b-tree was laid out on a virtual LUN taken from a DS4500 controller. The LUN was a RAID1 in a 2+2 pattern. Read and write caching on the DS4500 controller was disabled. This makes the controller behave much like a set of plain, basic disks. Reads go directly to disk, and writes are not completed until they reach the actual disk. This simplified the system under test.

The trees created in the experiments were spread across a 4GB area on disk. Table I shows the I/O performance of the disk subsystem for such an area. Three workloads were used: (1) read a random page, (2) write a random page, and (3) read and write a random page. When using a single thread, a 4KB write takes 17 milliseconds. A 4KB read takes about 4 milliseconds. Reading a random 4KB page and then writing it back to disk also takes 17 milliseconds. When using 10 threads, the throughput improves by a factor of 5 for reads and a factor of 6 for writes.

For the performance evaluation we used a large tree, T_{235}, with approximately 64000 leaves. The number of keys in a node is between b and $3b$. T_{235} has a maximal fanout of 235 entries and b is equal to $\frac{235}{3} = 78$. The depth of the tree is 4. The tree is created by randomly inserting and removing keys.

T_{235}.
Maximal fanout: 235
Legal no. entries: 78 .. 235
Contains: 9684662 keys and 65840 nodes (65345 leaves, 495 index-nodes)

Table II. Throughput Results (measured in
operations per second)

% in-memory	1 thread	10 threads	ideal
75	570	4394	4868
50	464	2271	2434
25	321	1508	1622
10	258	1254	1352
5	254	1227	1281
2	250	1145	1241

A pure random lookup-key workload is applied to T_{235}.

Tree depth is: 4
Root degree is: 4
Node average capacity: 148

A set of experiments starts by creating a base tree of a specific fanout and flushing it to disk. A special procedure is used. A clone q is made of the base tree. For read-only workloads, 1000 random lookup-key operations are performed. For other workloads the clone is aged by performing 1000 random insert-key/remove-key operations. Then, the actual workload is applied to q. At the end, the clone is deleted. This procedure ensures that the base tree, which took a very long time to create, isn't damaged and can be used for the next experiment. Each measurement is performed five times and results are averaged. The standard deviation for all the experiments reported here was 1% of the average, or less.

For each experiment the number of cache pages is fixed at initialization time to be some percentage of the total number of pages in the tree. This ratio is called the *in-memory* percentage. The in-memory ratio is the most important parameter in terms of performance.

We wanted to compare our b-tree structure to a bottom-up b-tree, or a b-link tree. However, such structures have no provision for strict shadowing, where every operation ripples to the top of the tree. Therefore, they would have to be locked in their entirety for each operation. This would not be a fair comparison. Instead, we compared our b-trees to an idealized data structure. An *ideal* data structure can locate leaf nodes without incurring the overheads of an indexing structure. This allows devoting the entire cache to leaf nodes. To compute ideal performance, we assumed that the CPU was infinitely fast.

7.1 Effect of the In-Memory Percentage on Performance

The in-memory percentage has a profound effect on performance. A pure random lookup-key workload was run against T_{235} with in-memory ratios 75%, 50%, 25%, 10%, 5%, and 2%. Each experiment included 20000 random lookup-key operations and throughput per second was calculated. If the in-memory percentage is x, then, under ideal performance, x of the workload is absorbed by the cache and the rest of the workload reaches the disk; throughput per second would be $1217 \times \frac{1}{1-x}$. Table II summarizes the results.

When the tree is not entirely in memory, we get close to ideal performance. Furthermore, throughput improvement is about x5, meaning that we

Table III. Throughput Results (measured in
operations per second)

% in-memory	1 thread	2 threads	ideal
100	14297	20198	N/A

A pure random lookup-key workload is applied to T_{235}, and
the entire tree is in memory.

Table IV. Latency for Single-Key Operations
(in milliseconds)

Lookup	Insert	Remove Key	Append
3.415	16.8	16.6	0.006

The in-memory percentage was 5%.

are making the most out of the available disk parallelism. The frequency of access to index-nodes is much higher than to leaf nodes; a good cache algorithm is able to keep the top levels of the tree in memory. If all index-nodes can be cached, then operations like lookup-/remove-/insert-key will access a single on-disk leaf page. Indeed, for all the cases shown in Table II all the index-nodes fit in memory. That is why there is little performance difference between 2%, 5%, 10% and 25%. The performance difference between 50% and 2% is only a factor of 2.

For the special case where the entire tree fits in memory, we used a thread per CPU and 10^8 operations. The results are depicted in Table III. Using more threads did not improve performance and sometimes reduced it.

In the rest of this section we choose 5% and 100% as the two representative in-memory ratios.

7.2 Latency

There are four operations whose latencies were measured: lookup-key, insert-key, remove-key, and append-key. In order to measure latency of operation x, an experiment was performed where x was executed 20000 times, and total elapsed time was measured. The in-memory percentage was 5% and the latency per operation was computed as the average. Operations were performed with randomly chosen keys.

Table IV shows the latency of the b-tree operations on the tree T_{235}. The cost of a lookup is close to that of a single disk-read. An insert-key requires reading a leaf from disk and modifying it. The dirty page is later flushed to disk. The average cost is therefore a disk-read and a disk-write, or, about 17 mili-seconds. The performance of remove-key is about the same as an insert-key; the algorithms are very similar. Append always costs 6 us because the pages it operates on are always cached.

7.3 Throughput

Throughput was measured using four workloads taken from Srinivasan and Carey [1993]: *Search-100*, *Search-80*, *Update*, and *Insert*. Each workload is a combination of single-key operations. *Search-100* is the most read intensive: It

Table V. The Four Different Workloads

	Lookup	Insert	Remove
Search-100	100%	0%	0%
Search-80	80%	10%	10%
Modify	20%	40%	40%
Insert	0%	100%	0%

Table VI. Throughput Results (measured in operations per second)

Tree	No. threads	Src-100	Src-80	Modify	Insert
T_{235}	10	1227	748	455	400
	1	272	144	75	62
Ideal		1281			429

performs 100% lookup. *Search-80* mixes some updates with the lookup workload; it performs 80% lookups, 10% remove-key, and 10% add-key. *Modify* is mostly an update workload; it performs 20% lookup, 40% remove-key, and 40% add-key. *Insert* is an update-only workload; it performs 100% insert-key. Table V summarizes the workloads.

Each operation was performed 20000 times and throughput per second was calculated. Five such experiments were performed and averaged. The throughput test compared running a workload using one thread compared with the same workload, but executed concurrently with ten threads. CPU utilization throughout all the tests was about 1%; the tests were all I/O bound.

Table VI shows ideal performance and the results for a single thread and for ten threads. The in-memory ratio was 5% and the tree was T_{235}. The throughput gain in all cases is roughly x5 to x6.

In the *Search-100* workload each lookup-key translates into a disk read for the leaf node. This means that ideal throughput is $1217 \times \frac{1}{0.95} = 1281$ requests per second. Actual performance is within 3% of the ideal.

In the *Insert* workload each insert-key request is translated, roughly, into a single disk-read and a single disk-write of a leaf. This means that ideal throughput is $408 \times \frac{1}{0.95} = 429$. Actual performance is within 7% of ideal.

The *Modify* and *Search-80* workloads perform somewhere in the middle of *Insert* and *Search-100*.

7.4 Performance Impact of Checkpoints

During a checkpoint all dirty pages must first be written to disk before they are reused. It is not possible to continue modifying a dirty page that is memory resident; it must be evicted to disk first in order to create a consistent checkpoint.

In terms of performance of an ongoing workload, the worst case occurs when all memory-resident pages are dirty at the beginning of a checkpoint. The best case occurs when all memory-resident pages are clean. Then, the checkpoint occurs immediately, at essentially no cost.

In order to assess performance, the throughput tests were run against T_{235}. After 20% of the workload was complete, that is, after 4000 operations, a

Table VII. Throughput Results When a Checkpoint is
Performed During the Workload

Tree	Src-100	Src-80	Modify	Insert
checkpoint	1217	734	430	379
base	1227	748	455	400

The in-memory percentage is 5%, the tree is T_{235}.

Table VIII. Throughput Results with T_{235} and Ten Threads

	Src-100	Src-80	Modify	Insert
2 clones	1211	709	423	367
base	1227	748	455	400

The in-memory percentage is 5%. Measurements are in operations per
second.

checkpoint was initiated. Table VII shows the performance for tree T_{235} with
10 threads. The first row shows results when running a checkpoint. The second
row shows base results, for ease of reference.

For the *Search-100* workload there was virtually no degradation. This is
because there were no dirty pages to destage. Other workloads suffered between
3% and 10% degradation in performance.

7.5 Performance for Clones

In order to assess the performance of cloning, a special test was performed.
Two clones of the base tree were created: p and q. Both clones were aged by
performing $\frac{1000}{2} = 500$ operations on them. Finally, $\frac{20000}{2} = 10000$ operations
were performed against each clone.

Table VIII shows performance for tree T_{235} with 10 threads when 5% of the
tree is in memory. The first row shows results with 2 clones. The second row
shows base results, for ease of reference.

There is little performance degradation when using clones. The clock-caching
algorithm is quite successful in placing the index-nodes for both clones into the
cache. This also shows that concurrency is good, even when using clones.

It is interesting to see what kind of concurrency is achieved when the two
clones are entirely in memory. Table IX shows throughput results when one or
two threads update the clones. We can see that the scaling is not linear; we do
not get a factor of 2 improvement in performance by using two threads. This
is because the locking overhead is significant, limiting the performance gains
from use of multiple CPUs.

We were interested in performance when many clones are used. We wanted a
workload with some locality to mimic expected usage patterns of cloned files and
volumes. We created 256 clones from the base tree and ran a special variation
of the SRC-80 benchmark against the clones. Each thread ran a main loop in
which it chooses an operation (80% lookup, 10% remove-key, 10% insert-key)
and chooses a random key k, and performed the operation serially against
keys $[k \ldots k + 99]$. No warmup of the clones was performed. Table X shows the
performance results.

Table IX. Throughput Results with Two Clones of T_{235}

No. threads	Src-100	Src-80	Modify	Insert
2	20395	18524	16907	16505
1	13910	12670	11452	11112

Both clones fit in memory, and the in-memory percentage is 100%.

Table X. Throughput Results with
256 Clones of T_{235}

Tree	No. threads	Src-80'
T_{235}	10	22198
	1	5008

The in-memory percentage is 5%. The benchmark is a modified SRC-80.

We can see that performance is comparable to in-memory performance. The cache and b-tree make good use of the locality in the workload.

8. FUTURE WORK

Several issues that can have a significant impact on performance have not been studied here:

—space allocation;

—write-batching; and

—more sophisticated caching algorithms, for example, ARC [Megiddo and Modha 2003].

These topics were not addressed not because they are unimportant, but rather because each deserves much wider treatment than we could hope to achieve here.

Delayed allocation allows coalescing dirty pages and writing them to disk in long extents. Such an optimization can significantly impact our performance results. However, with the advantage of allowing long disk writes comes the disadvantage of requiring defragmentation. Defragmentation can not only be costly, but also complicated in the presence of snapshots.

Caching has a major impact on b-tree performance. Under random workloads LFU is advantageous; under workloads with more locality LRU is advantageous. Furthermore, in a full-fledged file system, multiple b-trees from partially shared clones are accessed at the same time. Which pages should we cache?

We believe each of these issues merits further study.

9. SUMMARY

B-trees are an important data structure used in many file systems. B-trees are attractive because: (1) they provide guaranteed logarithmic lookup and update complexity; and (2) they have effective caching strategies.

Shadowing is a powerful technique for updating file system data-structures. It is used by file systems like ZFS and WAFL to guarantee operation atomicity

and system recoverability. Shadowing also allows optimizations such as write-coalescing and operations like cloning.

Using b-trees together with shadowing raises concurrency challenges, as well as problems due to leaf chaining. In this article we described a set of algorithms that overcome these obstacles. This allows building systems that enjoy the benefits of both algorithms. We have also shown how to clone a single file as well as a full volume in a file system.

ACKNOWLEDGMENTS

REFERENCES

BAYER, R. AND MCCREIGHT, E. 1972. Organization and maintenance of large ordered indices. *Acta Informatica*, 173–189.

BAYER, R. AND SCHKOLNICK, M. 1977. Concurrency of operations on B-trees. *Acta Informatica 9*, 1–21.

BEST, S. 2002. Journaling file systems. *Linux Mag*.

COMER, D. 1979. Ubiquitous B-tree. *ACM Comput. Surv. 11,* 2, 121–137.

GRAEFE, G. 2004. Write-Optimized B-trees. In *Proceedings of the International Conference on Very Large Database (VLDB)*, 672–683.

GRAY, J. AND REUTER, A. 1993. *Transaction Processing: Concepts and Techniques*. Mogran Kaufmann.

GUIBAS, L. AND SEDGEWICK, R. 1978. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*.

HENSON, V., AHRENS, M., AND BONWICK, J. 2003. Automatic performance tuning in the Zettabyte file system. In *USENIX Conference on File and Storage Technologies* (work in progress report).

HITZ, D., LAU, J., AND MALCOLM, M. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Technical Conference*.

LANIN, V. AND SHASHA, D. 1986. A symmetric concurrent B-tree algorithm. In *Proceedings of the ACM Fall Joint Computer Conference*, Dallas, TX, 380–389.

LEHMAN, P. AND YAO, S. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6*, 4, 650–670.

LOMET, D. 2001. The evolution of effective B-tree: Page organization and techniques: A personal account. *ACM SIGMOD Rec. 30*, 3(Sept.), 64–69.

LOMET, D. AND SALZBERG, B. 1992. Access method concurrency with recovery. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 351–360.

MCKUSICK, M., JOY, W., LEFFLER, S., AND FABRY, R. 1984. A fast file system for Unix. *ACM Trans. Comput. Syst*.

MEGIDDO, N. AND MODHA, D. S. 2003. ARC: A self-tuning, low overhead replacement cache. In *USENIX Conference on File and Storage Technologies (FAST)*.

MENON, J., PEASE, D., REES, R., DUYANOVICH, L., AND HILLSBERG, B. 2003. IBM storage tank a heterogeneous scalable SAN file system. *IBM Syst. J. 42*, 2, 250–267.

MOHAN, C. AND LEVINE, F. 1992. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 371–380.

MOND, Y. AND RAZ, Y. 1985. Concurrency control in B+ trees databases using preparatory operations. In *the 11th International Conference on Very Large Data Bases*.

REISER, H. 2004. ReiserFS. http://www.namesys.com/.

RODEH, O. 2006. B-trees, shadowing, and range-operations. Tech. Rep. H-248. November.

ROSENBERG, J., HENSKENS, F., BROWN, A., MORRISON, R., AND MUNRO, D. 1990. Stability in a persistent store based on a large virtual memory. *Secur. Persist.* 229–245.

SOULES, C., GOODSON, G., STRUNK, J., AND GANGER, G. 2003. Metadata efficiency in a comprehensive versioning file system. In *USENIX Conference on File and Storage Technologies (FAST)*.

SRINIVASAN, V. AND CAREY, M. 1993. Performance of B+ tree concurrency control algorithms. *VLDB J. 2*, 4, 361–406.

SWEENY, A., DOUCETTE, D., HU, W. , ANDERSON, C., NISHIMOTO, M., AND PECK, G. 1996. Scalability in the XFS file system. In *Proceedings of the USENIX Technical Conference*, 1–14.