H-0245 (H0611-006) November 12, 2006
Computer Science

IBM Research Report

B-trees, Shadowing, and Clones

Ohad Rodeh
IBM Research Division
Haifa Research Laboratory
Mt. Carmel 31905
Haifa, Israel

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

B-trees, Shadowing, and Clones

Ohad Rodeh, IBM Research

Abstract

B-trees are used by many file-systems to represent files and directories. They provide guarantied logarithmic time key-search, insert, and remove. Shadowing, or copy-on-write, is used by other file-systems 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 paper 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, basically unlimited, number of clones. These algorithms were used in an experimental object-disk.

We believe that this work is applicable not only to object-disks but also to other file-systems.

1 Introduction

B-trees [18] are used by several file systems [3, 20, 10, 9] to represent files and directories. Compared to traditional i-nodes [14] b-trees offer guaranteed logarithmic time key-search, insert and remove. Furthermore, b-trees can represent sparse files well.

Shadowing is a technique used by some file systems to ensure atomic update to persistent data-structures [2, 6, 11, 15, 22]. 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 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, 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. 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 on be deallocated.

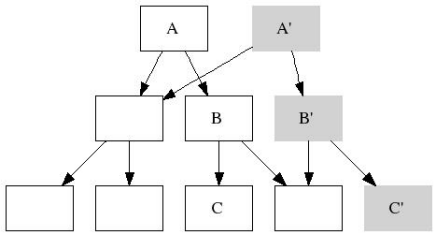


Figure 1: Modifying a leaf requires shadowing up to the root.

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 the deletion of the snapshot would it be deallocated. The upshot is that pages can be shared between many snapshots; indeed, whole subtrees can be shared between snapshots.

This work was performed as part of research into building an object-disk storage device (OSD) [21, 4]. Roughly speaking, an OSD is a primitive file system that is exported through a network interface using a standardized protocol. It was desirable to (1) use b-trees to implement the persistent OSD data-structures and (2) use shadowing for update and snapshots. This would combine the good properties of both techniques: logarithmic access data-structures coupled with simple logging, crash-recovery, and snapshots. However, we ran into serious difficulties when trying to use these techniques together.

The classic persistent recoverable b-tree, as described in the literature [5, 8], is updated using a bottom-up 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 re-balancing operations and range lookups. There are good concurrency schemes allowing multiple threads to update a tree; the best one is currently b-link trees [17].

The main issues when trying to apply shadowing to the classic b-tree are:

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 copy-on-write leaves cannot be linked together. For example, Figure 2 shows a tree whose right most 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.

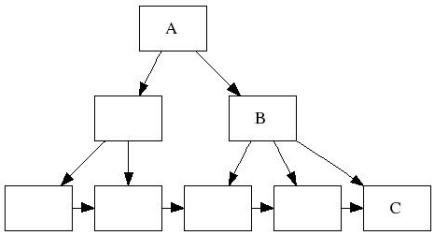


Figure 2: A tree whose leaves are chained together. The right most leaf is C and B is its immediate ancestor. If C is modified, the entire tree has to be shadowed.

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 re-balancing purposes after a remove-key operation. When using copy-on-write this habit could be expensive. For example, in Figure 3 a 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.

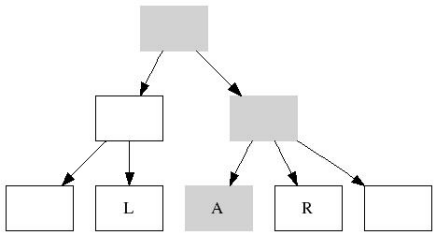


Figure 3: Removing a key and effects of re-balancing and shadowing.

This work describes the first b-tree construction that can coexist with shadowing while providing good concurrency. This is a fundamental 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.

The rest of this paper is organized as follows: Section 2 is about related work, Section 3 described the experimental object-disk, Section 4 discusses recoverability, Section 5 describes the basic algorithms, Section 6 describes cloning, Section 7 describes the run-time system, Section 8 shows performance, Section 9 discusses future work, and Section 10 summarizes.

2 Related work

There is an alternate style of copy-on-write used in some databases [12] that is beyond the scope of this paper. The only form of shadowing discussed here is the one described in Section 1.

There are few papers that discuss concurrency together with recoverability of b-trees, see [5, 8]. The challenge in constructing an algorithm that achieves both goals is that recoverability severely constrains concurrency. Furthermore, we found no published papers on concurrency, recoverability, and b-trees with the added constraint of shadowing.

This work makes use of top-down b-trees; these were first described in [13], [25], and [23].

Some file-systems use b-trees to represent directories [3, 20, 9]. One of the difficulties in handling directory entries is that, unlike b-tree values in the OSD, they are variable size. We believe that the b-trees described here can be adapted to handle variable size values.

It is possible to use disk-extents instead of fixed-sized blocks [3, 20, 9]. We have experimented with using extents in the OSD. The resulting algorithms are similar in flavor to those reported here and we do not describe them for brevity.

The WAFL system [6] 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 two main limitations which we improve upon:

1. WAFL is limited to 32 snapshots.

2. In WAFL the free-space bits of all blocks that belong to a newly created snapshot need to be set to 1 as part of snapshot creation.

In our algorithm

1. The limit is $2^{32}-1$ for the same space cost as WAFL

2. Only the children of the root of the clone are involved in snapshot creation. Free-space map operations are performed gradually through the lifetime of the clone.

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

3 Object-disk (OSD)

An object-disk, according to the SNIA/T10 specification [21, 4], is essentially a primitive file system that is exported through a network interface using a standardized protocol. The OSD contains objects, which have 64-bit names, and an object-catalog that indexes them. There are no directories in an OSD. Objects are much like files in a regular file system except that they are likely to be sparse. The important commands are: create, delete, read, and write object. Snapshots are also supported. There is a command that creates a writeable snapshot of the OSD object-system.

Our group built an experimental object-disk. The two main types of persistent data-structures in our OSD are objects and the object-catalog. B-trees were a natural fit for these two data structures.

The design point for the OSD was that it would be part of a storage controller. Such systems typically have severe limits on memory and CPU cycles. Therefore, It was important to try to maintain a small footprint. Using a generic database with full fledged transactions was not possible.

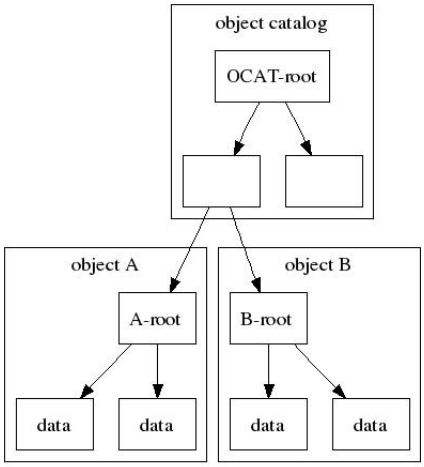


Figure 4: An object-catalog that points to objects A and B.

Space is managed in 4KB blocks, also called pages. Disk addresses are represented by 64-bits to accommodate large disks. An object contains data blocks that are aggregated using a b-tree. The b-tree maps offsets in the object to data pages on disk. The object-catalog (OCAT) maps an object-id to the root-page of the b-tree for that object; the OCAT is also a b-tree. Figure 4 shows an example where the catalog points to two objects: A and B. The b-tree index nodes are laid out on 4KB meta-data pages.

The b-trees representing objects and the catalog both map 64-bit keys to 64-bit values. More generally, fixed-sized keys to fixed-sized values.

Section 5 describes the b-tree algorithms for create, delete, lookup-key, remove-key, and insert-key. Here, we show how several object-disk commands are mapped to these operations.

A create-object(B) command is implemented by:

```
* B-root = lookup-key(OCAT-root, B)
* if (B-root != 0)
    - return obj-already-exists
  else
    - B-root = create-tree()
    - insert-key(OCAT, B, B-root)
```

A delete-object(B) command is implemented by:

```
* B-root = lookup-key(OCAT-root, B)
* if (B-root == 0)
    - return obj-does-not-exist
  else
    - remove-key(OCAT-root, B)
    - delete-tree(B-root)
```

A read(*B, ofs, len=4KB*) command, when the offset is 4KB aligned, is implemented by:

```
* B-root = lookup-key(OCAT-root, B)
* if (B-root == 0)
    - return obj-does-not-exist
  else
    - addr = lookup-key(B-root, ofs)
    - if (addr == 0) data=zeros
    - otherwise, read data from addr
    - send data to client
```

A write(*B, ofs, len=4KB, data*) command, when the offset is 4KB aligned, is implemented by:

```
* B-root = Lookup-key(OCAT-root, B)
* if (B-root == 0)
    - return obj-does-not-exist
  else
    - addr = allocate 4KB on disk
    - write data to disk at addr
    - insert-key(B-root, ofs, addr)
```

In order to support writable snapshots on the OSD we chose to support cloning at the b-tree level. To *clone* a b-tree means to create a writable copy of it that allows all operations: lookup, insert, remove, and delete. This is described in Section 6.

The expected workload is mostly read/write commands; create/delete commands are less frequent. Clone commands are assumed to be infrequent. In terms of b-tree operations this translates into a high frequency of lookup-key/insert-key and a low frequency of remove-key/create/delete.

The OSD handles multiple commands concurrently. When a command arrives, the runtime system checks if there are enough resources to execute it. If so, the resources are reserved and the command is logged and executed; otherwise, the command is rejected. In order to simplify the runtime system we chose not to abort nor roll back commands. Therefore, it is crucial to compute in advance a worst-case estimate on the command's resource-usage and to practice deadlock avoidance. The two important resources are memory-pages and disk-pages. The amount of memory, depending on configuration, can be very low, so memory-usage of each command should be low.

This design means that the b-tree implementation has to:

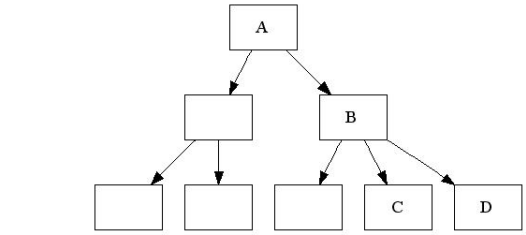
1. Have good concurrency
2. Work well with shadowing
3. Use deadlock avoidance
4. Have guarantied bounds on disk-space and number of memory-pages required per each b-tree operation

Section 5 goes into how such a b-tree is constructed.

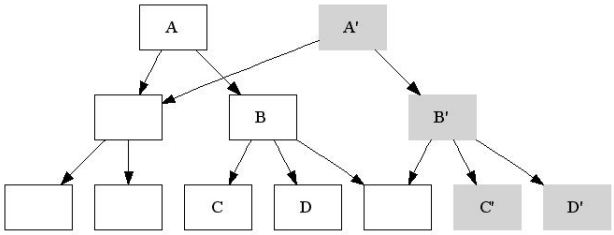
4 Recoverability

Shadowing file systems 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.

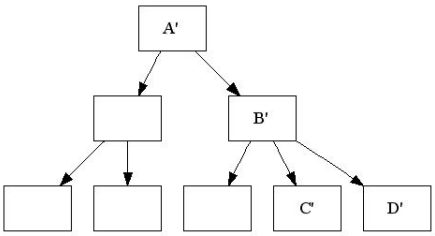
For example, Figure 5(a) shows an initial tree. Figure 5(b) shows a set of modifications marked in gray. Figure 5(c) shows the situation after the checkpoint has been committed and unreferenced pages have been deallocated.



(a) Initial file system tree



(b) Modifications



(c) New checkpoint

Figure 5: Checkpoints.

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 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.

The OSD implements cloning. Checkpoints are simply clones that users cannot access. Clones are described in Section 6.

5 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, index nodes contain mappings between keys and child nodes; see Figure 6. 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 [7]. There are no links between leaves.

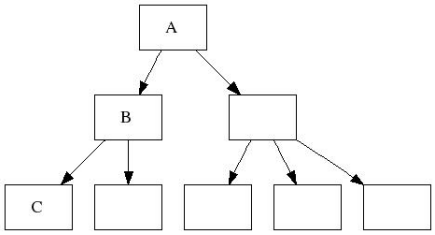


Figure 6: Three types of nodes in a tree; A is a root node, B is an index node, and C is a leaf node. Leaves are not linked together.

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 7 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 $2b+1$ 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 $[b \ldots 2b+1]$.

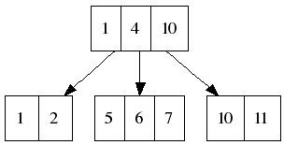


Figure 7: A b-tree with two levels.

A pro-active 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. Pro-active 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* [19] is used. Lock coupling (*or crabbing*) is locking children before unlocking the parent. This ensures the validity of a tree-path that a task is traversing without pre-locking 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.

5.1 Create

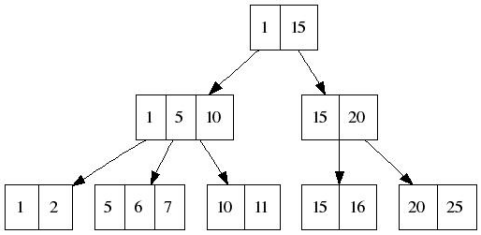
In order to create a new b-tree a root page is allocated and formatted. The root page is special, it can contain zero to $2b+1$ entries. All other nodes have to contain at least b entries. Figure 8 presents a tree that contains a root node with 2 entries.

Figure 8: A b-tree containing only a root.

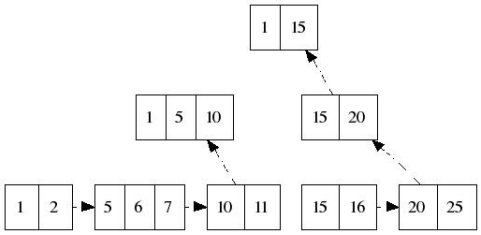
5.2 Delete

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

An example for the post-order delete pass is shown in Figure 9. A tree with eight nodes is deleted.



(a) Tree to be deleted



(b) A post-order traversal

Figure 9: Deleting a tree.

5.3 Insert key

Insert-key is implemented with a pro-active 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.

Figure 10 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.

5.4 Lookup key

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

5.5 Remove key

Remove-key is implemented with a pro-active merge policy. On the way down to a leaf each node with a minimal amount of keys is fixed, making sure it will have

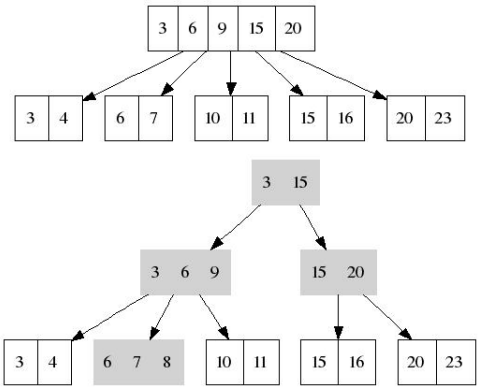


Figure 10: Inserting key 8 into a tree. Gray nodes have been shadowed. The root node has been split and a level was added to the tree.

at least $b + 1$ keys. This guaranties that removing a key from the leaf will, at worst, effect its immediate ancestor. During the descent in the tree lock-coupling is used. Locks are taken in exclusive mode.

For example, Figure 11 shows a remove-key operation that fixes index-node $[3, 6]$ by merging it with its sibling $[15, 20]$.

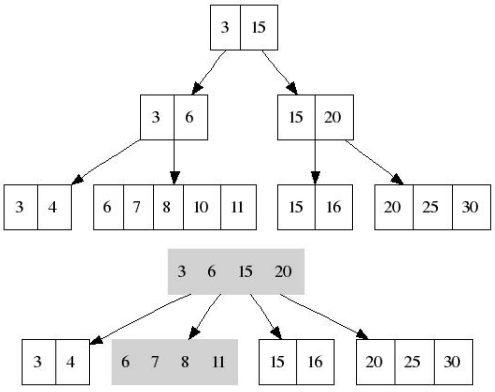


Figure 11: Removing key 10 from a tree. Gray nodes have been shadowed. The two children of the root were merged and the root node was replaced.

5.6 Resource analysis

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

5.7 Comparison to standard b-tree

A top-down b-tree with bounds $[b, 2b + 1]$ is 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 of 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 $2b$.

6 Clones

This section builds upon Section 5 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:

- 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 does not provide 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 maps 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, modified nodes are colored in light gray. In order to better visualize the algorithms reference counters are drawn inside nodes. This can be misleading, the ref-counters are physically located in the free-space maps.

6.1 Create

The algorithm for cloning a tree T_p is:

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 12. Tree T_p contains seven nodes and T_q is created as a clone of T_p by copying the root P to Q. Both roots point to the shared children: B and C. The reference counters for B and C are incremented to 2.

Notice that in Figure 12(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.

6.2 Lookup

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

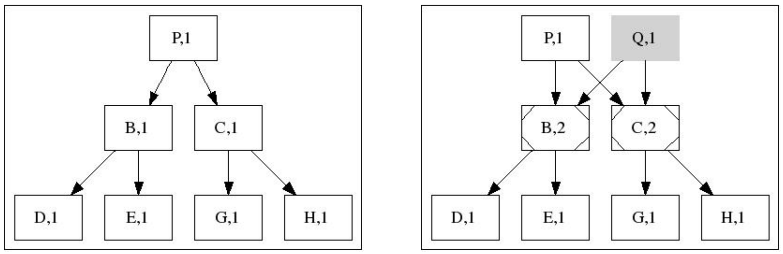
6.3 Insert-key and Remove-key

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 run-time system know that the page is about to be modified and gives it a chance to shadow the page if necessary.

Without clones, the only requirement for the mark-dirty operation is to check if the page does not belong to the previous checkpoint; if so, the page must be shadowed. Otherwise, it can be modified in place. With clones, this is more subtle. 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.



(I) Initial tree T_p (II) Creating a clone T_q

Figure 12: Cloning a b-tree.

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 13 shows an example of a two trees, T_p and T_q, that start out sharing all their nodes except the root. Initially, all nodes are clean. A key is inserted into leaf node 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 one, so nothing special is needed. In stage (III) node C is shadowed, this splits C into two versions, one belonging to T_p the other to T_q each with a ref-count of 1. The children of C are nodes H and G, their ref-count is incremented to two. In stage (IV) node H is shadowed, this splits H into two separate versions each with ref-count 1.

Performing the mark-dirty in this fashion allows delaying the ref-count operations. For example, in Figure 13(I) node C starts out with a ref-count of two. 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 two 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.

6.4 Delete

The delete algorithm is also affected by the free-space ref-counts. Without cloning, a post-order 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 post-order traversal node N is reached:

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 one then it belongs only to T_p. Continue downward traversal and on the way back up deallocate N.

Figure 14 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.

6.5 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 a potential of significantly impacting performance.

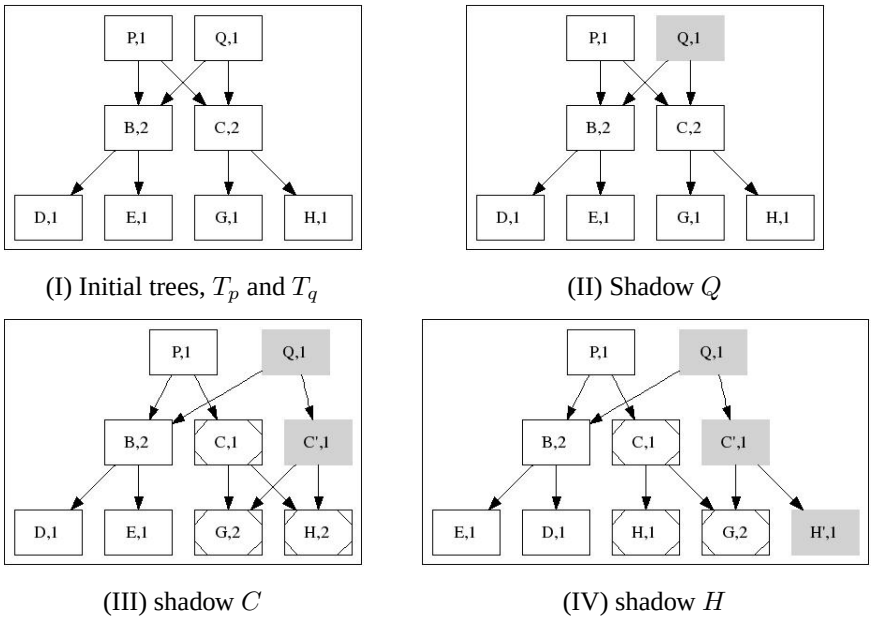
Several observations make this unlikely:

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

- If a page is dirty and remains in-memory, no additional checking is needed.

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

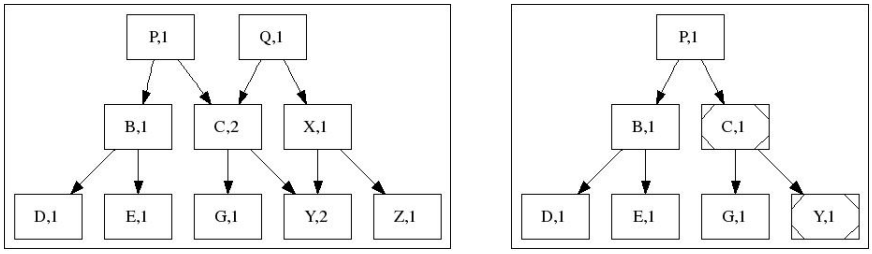
A major cost to free-space counters is the increased size of free-space map. Instead of keeping a bit per block like most file systems, a counter is needed. If 32-bit counters are used then the map grows by a factor of 32. This also allows supporting up to 2^{32} clones. The WAFL file system [6] uses 32-bits in its free-space map and it is reputed to have to good performance. This gives the author reason to believe that this issue can be negotiated.



(I) Initial trees, T_p and T_q (II) Shadow Q

(III) shadow C (IV) shadow H

Figure 13: Inserting into a leaf node breaks sharing across the entire path.



(I) Initial trees T_p and T_q (II) After deleting T_q

Figure 14: Deleting a b-tree rooted at Q.

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.

7 The run-time system

A minimal run-time 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 log-structured 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 clean and dirty pages. A simple clock scheme is implemented, no attempt is made to coalesce pages written to disk into large writes, no pre-fetching 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 the old checkpoint, 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. There is a ref-count per block. This was done to eliminate any noise generated by the particulars of the OSD free-space component.

A log was not used, it is assumed that the OSD protects all b-tree operations through logical logging of commands.

A special threading package was used, it is similar to [1]. The idea is to use a single operating-system thread, the *main-thread*, to run all the complex code: caching, free-space, b-tree, command logic, etc. Separate operating-system threads perform the heavy lifting: networking and IO. The main-thread executes multiple light-weight *tasks*. Tasks are much like regular threads except that they are non-preemptive and they cannot perform regular system-calls. A task yields the CPU either voluntarily or when it performs an IO. In the experimental setup for this work most of the OSD code has been eliminated; the upshot is that only the main-thread is executed along with the IO threads. This limits any b-tree code to execute on a single CPU. While the b-tree algorithms themselves are thread-safe for any threading package, they are limited here to execute on a single CPU.

This system does not contain any kernel code. It was built and tested on a Linux operating system with an Intel processor.

8 Performance

The OSD was built to be part of a storage controller. It was specified to be able to manage terrabytes of disk space with gigabytes of memory. Most of the memory was to be used for caching customer data, most of the CPU cycles were to be spent on networking and IO. The b-tree was assumed to reside mostly on disk, with frequently accessed pages in memory. The b-tree code was to use little CPU.

In order to achieve good performance the b-tree had to:

1. Work well when most of the tree is not in-memory
2. Use little CPU
3. Get good concurrency from the disk subsystem

In this section we show that the algorithms, indeed, achieve these goals.

In [24] there was 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. This creates a serialization point. We found that 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.

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

The test-bed used in the experiments was a single machine connected to a DS4400 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 DS4400 controller. The LUN is a RAID5 in an 8+P pattern. Strip width is 64KB, this means that full stripe is $8 \times 64KB = 512KB$. Read and write caching is disabled.

The trees created in the experiments were spread across a 1GB area on disk. Table 1 shows the IO-performance of the disk subsystem for such an area. Three workloads were used (1) read a random page (2) write a random page (3) read and write a random page. When using a single thread a 4KB write takes 18 milliseconds, this is due to the RAID-5 penalty for short writes. A short write requires 2 reads and 2 writes. A 4KB Read takes about 5 milliseconds. Reading a random 4KB page and then writing it back to disk takes 24 milliseconds. When using 10 threads throughput improves by a factor of six.

#threads	op.	time per op.(ms)	ops per second
10	read	N/A	1283
	write	N/A	421
	R+W	N/A	311
1	read	4.8	207
	write	18.3	68
	R+W	24.6	41

Table 1: Basic disk-subsystem capabilities. Three workloads were used (1) read a random page (2) write a random page (3) read and write a random page. Using 10 threads increases the number of operations per second by a factor of six.

Therefore, large trees with about 64,000 leaves were used to empirically assess performance. It turned out that the only way to quickly build such large trees was through an append only workload. The even numbers $\{0,2,4, \ldots\}$ were chosen as keys; they were inserted sequentially into the tree.

Two base-trees were used T_{235} and T_{150}. 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 $\frac{235}{3} = 78$. T_{150} has a maximal fanout of 150 and b is equal to $\frac{150}{3} = 50$. A node can hold more than 150 entries; therefore, this limit is artificially enforced by wasting some of the space in a page.

T_{235}

Maximal fanout: 235
Legal #entries: 78 .. 235
Contains: 7520000 keys and 64827 nodes (64273 leaves, 554 index-nodes)
Tree depth is: 4
Root degree is: 4
Index node average fanout: 117
Leaf node average capacity: 117

T_{150}

Maximal fanout: 150
Legal #entries: 50 .. 150
Contains: 4800000 keys and 64864 nodes (63999 leaves, 865 index-nodes) Tree depth is: 4
Root degree is: 11
Index node average fanout: 75
Leaf capacity average capacity: 75.00

T_{235} is representative of the OSD catalog. T_{150} is representative of a tree where the key-value pairs take up 20bytes instead of 16bytes. This is an approximation of a tree that holds disk-extents. Both T_{235} and T_{150} have an average occupancy of 50%. This is caused by the append-only workload used to create them. When using append, the right edge of the tree keeps splitting leaving behind half-full nodes.

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.

Our b-tree construction is novel and there are no existing data-structures to compare it against. Therefore, we compare it to *ideal* performance that could be achieved with a data-structure the could somehow locate leaf nodes without incurring the overheads of an indexing structure. This would allow devoting the entire cache to leaf nodes. To compute ideal performance we assumed that the CPU was infinitely fast.

8.1 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 100%, 50%, 10%, 5% and 2%. Each experiment included 30000 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 workload reaches the disk; throughput per second would be $1283 \times \frac{1}{1-x}$. Table 2 summarizes the results.

Tree	% in-memory	1 task	10 tasks	ideal
T_{235}	100	91354	91705	∞
	50	393	2431	2566
	10	219	1374	1425
	5	207	1306	1350
	2	197	1230	1309

Table 2: Throughput results, measured in operations per second. A pure random lookup-key workload is applied to T_{235}.

When the entire tree is in memory there is no difference in performance between ten tasks and one. This is because all tasks share a single CPU, and it is 100% utilized. When memory percentages drop, the disk-parallelism comes into play. For the other percentages a speedup of about x6 is achieved.

Performance with 10 tasks is very close to ideal performance, except for the case where the entire tree is in-memory. There, it is hard to compete with an infinitely fast CPU.

Performance is logarithmic with respect to cache size. This is because the clock algorithm is able to keep all the index nodes for T_{235} in memory. This means that operations like lookup/remove/insert-key access, in most cases, one on-disk leaf page.

Performance differences between 10%, 5%, and 2% were very small, therefore, for the rest of the experiments we focused on the 5% case.

8.2 Latency

There are four operations whose latency was 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 30000 times, and total elapsed time was measured. The latency per operation was computed as the average. Operations were performed with randomly chosen keys.

Table 3 shows the latency of the b-tree operations on the two trees. The cost of a lookup is close to the cost 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 24ms. The performance of remove-key is about the same as an insert-key; the algorithms are very similar. Append always costs 12us because the pages it operates on are always cached.

Tree	Lookup	Insert	Remove-key	Append
T_{235}	4.780	24.175	24.437	0.012
T_{150}	4.839	24.567	24.372	0.012

Table 3: Latency for single-key operations in milliseconds.

8.3 Throughput

Throughput was measured using four workloads taken from [24], *Search-100*, *Search-80*, *Update*, and *Insert*. Each workload is a combination of single-key operations. *Search-100* is the most read-intensive, it performs 100% lookup. *Search-80* mixes some updates with the lookup workload; it performs 80% lookups, 10% remove-key, and 10% add-key. *Update* is an update mostly workload; it performs 20% lookup, 40% remove-key, and 40% add-key. *Insert* is an update-only workload; it performs 100% insert-key. Table 4 summarizes the workloads.

	lookup	insert	remove
Search-100	100%	0%	0%
Search-80	80%	10%	10%
Modify	20%	40%	40%
Insert	0%	100%	0%

Table 4: The four different workloads.

Each operation was performed 30000 times and throughput per second was calculated. Five such experiments were performed and averaged. The throughput test compared running a workload using one task compared with the same workload but executed concurrently with ten tasks. CPU utilization throughout all the tests was about 1%; the tests were all IO bound.

Table 5 shows ideal performance and the results for a single task and for ten tasks. There is little difference in performance between T_{235} and T_{150} this is because the caching algorithm is able to place all the index nodes in cache. The throughput gain in all cases is x6 or slightly better.

In the *Search-100* workload each lookup-key translates into a disk-read for the leaf node. This means that ideal throughput is $1283 \times \frac{1}{0.95} = 1350$ requests per second. Actual performance is within 3% of 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 $311 \times \frac{1}{0.95} = 327$. Surprisingly, actual performance exceeds ideal performance by about 10%. This is because we are using a write-back cache. After each experiment about 2000 dirty leaf nodes remain in cache and the cost of writing them to disk is not accounted for. This unfairly disadvantages the computation of ideal performance.

The *Modify* and *Search-80* workloads are somewhere in the middle between *Insert* and *Search-100*. Overall, the b-tree performs no worse than 4% less than ideal.

Tree	#tasks	Src-100	Src-80	Modify	Insert
T_{235}	10	1307	763	407	359
	1	209	104	47	41
T_{150}	10	1284	752	407	357
	1	206	102	47	40
Ideal		1350	798	384	327

Table 5: Throughput results, measured in operations per second.

8.4 Append

The performance of append has very different characteristics than performance of other operations. It is instructive to examine a 100% append workload. The base trees, T_{235} and T_{150}, were built using a single task that appended to them. The time to create the trees and the throughput in append operations/second is shown in Table 6. The in-memory percentage was 5%

Tree	#keys	Total time (sec)	append ops/sec
T_{235}	7520000	1565.1	4800
T_{150}	4800000	1564.8	3069

Table 6: Append throughput results when building trees T_{235} and T_{150}.

These throughput numbers are higher by two orders of magnitude compared with other workloads with a single task. This is because append has very good locality, it needs only the nodes at the right edge of the tree. If they are all in-memory then append can be performed at CPU speed. Once in a while, a split is needed which requires, in most cases, one additional page. Overall, there are very few IOs needed to perform this workload.

8.5 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 6000 operations, a checkpoint was initiated. Table 7 shows performance for tree T_{235} with 10 tasks. 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 are no dirty-pages to destage. Other workloads suffer between 3% and 10% degradation in performance.

Tree	Src-100	Src-80	Modify	Insert
checkpoint	1302	697	388	346
base	1307	763	407	359

Table 7: Throughput results, when a checkpoint is performed during the workload. The in-memory percentage is 5%, the tree is T_{235}.

8.6 Performance for clones

In order to assess the performance of cloning a special test was performed. Two clones of the base tree are created, p and q. Both clones are aged by performing $\frac{1000}{2} = 500$ operations on them. Finally, $\frac{30000}{2} = 15000$ operations are performed against each clone.

Table 8 shows performance for tree T_{235} with 10 tasks. The first row shows results with 2 clones. The second row shows base results, for ease of reference.

	Src-100	Src-80	Modify	Insert
2 clones	1303	733	394	350
base	1307	763	407	359

Table 8: Throughput results with T_{235} and ten tasks. The in-memory percentage is 5%. Measurements are in operations per second.

There is little performance degradation when using clones. The clock 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.

9 Future work

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

- Space allocation
- Write-batching
- More sophisticated caching algorithms, for example, ARC [16]

We believe each of this issues merits further study.

10 Summary

B-trees are an important data-structure used in many file-systems. Shadowing is a powerful technique for updating file-system data-structures.

This paper has shown how to use shadowing to update b-trees and get the benefits of both algorithms: snapshots, recoverability, concurrency, and logarithmic lookup and update. The algorithms are efficient and they make good use of the disk subsystem.

Although our testbed was an object-disk we believe the ideas are applicable to other file-systems.

References

[1] A. Adya, J. Howell, M. Theimer, W. Bolosky, and J. Douceur. Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming. In *Usenix Annual Technical Conference*, June 2002.

[2] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance under Unix. In *ACM Trans. Computer Systems*, February 1989.

[3] A. Sweeny, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS File System. In *USENIX*, 1996.

[4] SNIA Storage Networking Industry Association. OSD: Object Based Storage Devices Technical Work Group. http://www.snia.org/tech_activities/workgroups/osd/.

[5] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In *ACM SIGMOD international conference on Management of data*, pages 371 – 380, 1992.

[6] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *USENIX*, 1994.

[7] D. Lomet. The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. In *SIGMOD Record*, 2001.

[8] D. Lomet and B. Salzberg. Access method Concurrency with Recovery. In *ACM SIGMOD international conference on Management of data*, pages 351 – 360, 1992.

[9] H. Reiser. ReiserFS. `http://www.namesys.com/`.

[10] J. Menon, D. Pease, R. Rees, L. Duyanovich, and B. Hillsberg. IBM Storage Tank a Heterogeneous Scalable SAN File-System. *IBM Systems Journal*, 42(2):250–267, 2003.

[11] J. Ousterhout and F. Douglis. Beating the I/O Bottleneck: A Case for Log-Structured File Systems. In *ACM SIGOPS*, January 1989.

[12] J. Rosenberg, F. Henskens, A. Brown, R. Morrison, and D. Munro. Stability in a Persistent Store Based on a Large Virtual Memory. *Security and Persistence*, pages 229–245, 1990.

[13] L. Guibas and R. Sedgewick. A Dichromatic Framework for Balanced Trees. In *Nineteenth Annual Symposium on Foundations of Computer Science*, 1978.

[14] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 1984.

[15] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[16] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *USENIX File and Storage Technologies (FAST)*, March 2003.

[17] P. Lehman and S. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[18] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, pages 173–189, 1972.

[19] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9:1–21, 1977.

[20] S. Best. Journaling File Systems. *Linux Magazine*, October 2002.

[21] Object Based Storage Devices Command Set (OSD). http://www.t10.org/drafts.htm. T10 Working draft.

[22] V. Henson, M. Ahrens, and J. Bonwick. Automatic Performance Tuning in the Zettabyte File System. In *File and Storage Technologies (FAST), work in progress report*, 2003.

[23] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *Fall Joint Computer Conference*, 1986.

[24] V. Srinivasan and M. Carey. Performance of b+ tree concurrency control algorithms. *VLDB Journal, The International Journal on Very Large Data Bases*, 2 (4):361 – 406, January 1993.

[25] Y. Mond and Y. Raz. Concurrency Control in B+-trees Databases Using Preparatory Operations. In *Eleventh International Conference on Very Large Data Bases*, 1985.