

IBM Research Report

B-trees, Shadowing, and Range-Operations

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 Range-operations

Ohad Rodeh, IBM Research

Abstract

B-trees are used by many file-systems to represent files and directories. They provide guaranteed 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 respect shadowing and achieve good concurrency. 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 [15] 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, 16, 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.

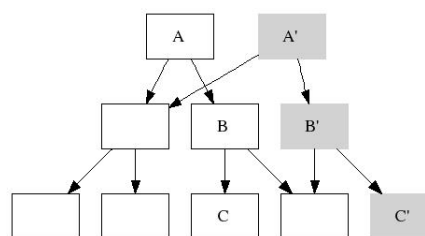


Figure 1: Modifying a leaf requires shadowing up to the root.

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 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.

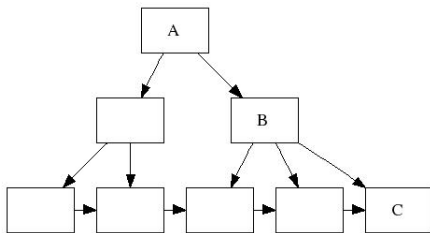


Figure 2: A tree whose leaves are chained together. The right most leaf is C and B is its immediate ancestor. If C is modified, the entire tree has to be shadowed.

Without links between leaves much of the b-tree literature becomes invalid. For example [5, 17] use leaf chaining to perform node split and merge. Furthermore, range operations become more difficult.

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.

This work describes and evaluates a special b-tree construction that can coexist with shadowing. The rest of

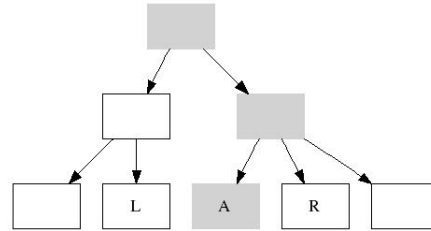


Figure 3: Removing a key and effects of re-balancing and shadowing.

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

2 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/write object, and truncate or clear a range in an 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.

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

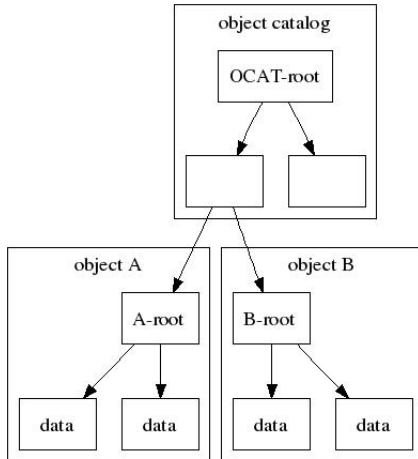


Figure 4: An object-catalog that points to objects *A* and *B*.

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, lookup-range, and remove-range. 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*) command is implemented by:

```
* B-root = lookup-key(OCAT-root, B)
* if (B-root == 0)
  - return obj-does-not-exist
else
  - Addr-list = lookup-range(
    B-root, ofs, len)
  - read data from Addr-list
  - send data to client
```

A remove-range(*B*, *ofs*, *len*) command is implemented by:

```
* B-root = Lookup-key(OCAT-root, B)
* if (B-root == 0)
  - return obj-does-not-exist
else
  - remove-range(B-root, ofs, len)
```

The expected workload is mostly read/write/ create/delete commands, truncate/clear commands are less frequent. In terms of b-tree operations this translates into a high frequency of lookup-key/insert-key/lookup-range/insert-range and a low frequency of remove-range.

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 guaranteed 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.

3 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.

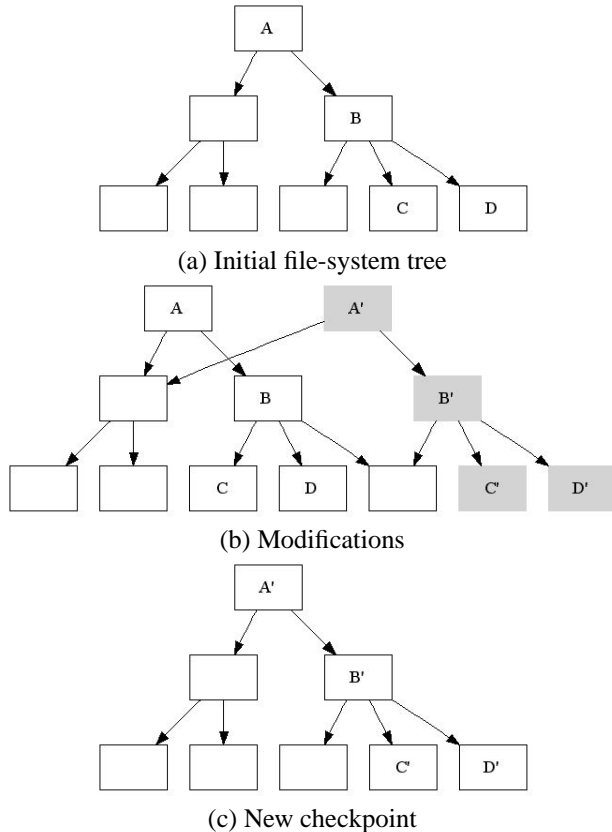


Figure 5: Checkpoints.

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.

4 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.

There are few papers describing a remove-range operation on a b-tree. The algorithm described here is based on the delete algorithm from [14]. The original paper does not handle shadowing and uses many more memory-pages than the solution described here.

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.

5 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.

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

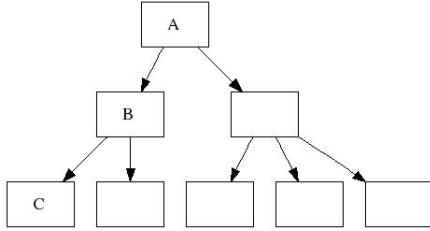


Figure 6: Three types of nodes in a tree; A is a root node, B is an index node, and C is a leaf node. Leaves are not linked together.

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.

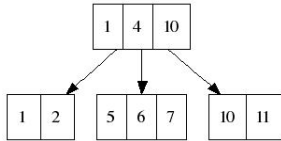


Figure 7: A b-tree with two levels.

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 $[b \dots 2b + 1]$.

A pro-active approach 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 constraints 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 $[2 \dots 5]$.

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.

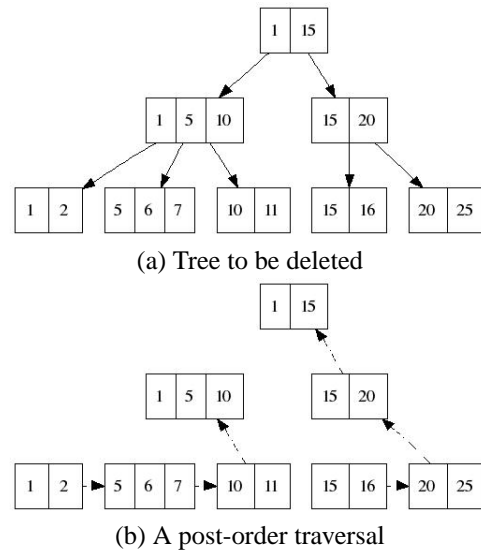


Figure 9: Deleting a tree.

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.

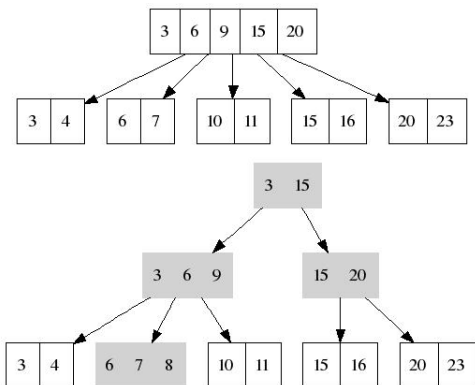


Figure 10: Inserting key 8 into a tree. Gray nodes have been shadowed. The root node has been split and a level was added to the tree.

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 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 $[15, 20]$.

5.6 Lookup range

A lookup range operation takes minimum and maximum key (\min , \max) values and returns the list of key-data

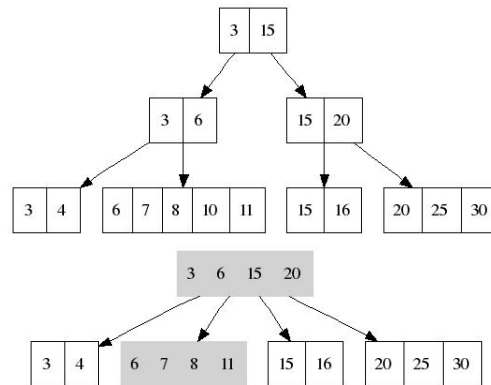


Figure 11: Removing key 10 from a tree. Gray nodes have been shadowed. The two children of the root were merged and the root node was replaced.

pairs in the range. This is problematic because:

- Leaves are not linked. With links, the algorithm would be (1) find a leaf containing \min and (2) traverse the tree to the right until finding \max .
- We do not want to lock the whole tree during the operation.

The semantics required from the algorithm are not very strong. Key-data pairs that are in the range for the duration of the operation must be returned. Keys that are concurrently added/removed may or may not be returned in the result list. These semantics are sufficient for the OSD.

The basic algorithm is:

```

* Start with cursor [min]
* On each iteration:
  - Descend through the tree
  - Read a leaf node
  - Set the cursor to the largest element in the leaf+1
  - return the matching elements in the leaf
* Perform iterations until reaching [max]

```

Descending through the tree requires using two pages and lock-coupling in shared mode.

Figure 12 shows an example where $\text{lookup}(6,120)$ is performed on a tree which contains elements between 3 and 128. The path taken by each iteration is marked in gray. The first iteration finds values $\{6,7,8\}$, the second iteration finds values $\{10,11\}$. A problem occurs in the third iteration which looks for value 12. Value 12 does not exist in the tree, how shall we proceed? Had

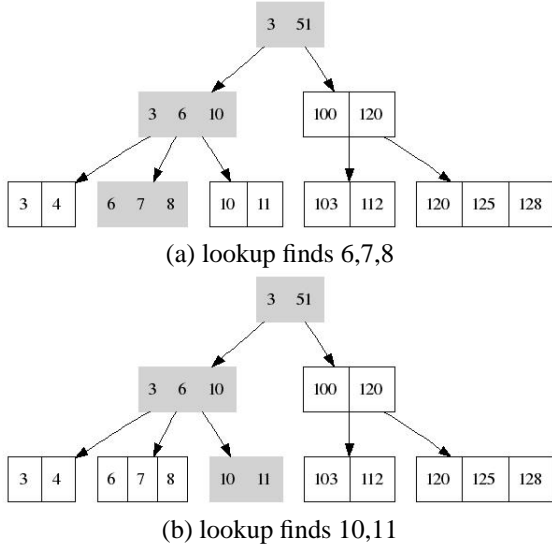


Figure 12: Lookup range [6,120]

leaves been chained together it would have been possible to proceed from node [10,11] directly to [103,112]. The alternative chosen here is to identify that value 12 lies on the boundary of the branch that [3] points to and to lock in shared-mode node [100,120] while searching in branch [3,6,10]. If entries are not found, then the search continues from [100,120].

This idea of identifying corner branch cases can be generalized, although we lack the space to fully describe the general solution here. In terms of resources three memory-pages and shared-mode locks are needed.

5.7 Remove-range

Remove range takes a range $ofs - end$ and removes all the keys inside it from the tree. The whole tree is locked during the operation. The justification is that removing an object range is assumed to be infrequent.

It is important to modify a small, bounded, number of tree pages. This improves sharing with old snapshots and makes it possible to reserve sufficient resources in advance. It is crucial to restore the b-tree invariants because that is the foundation of the resource estimates.

A trivial approach is to iterate on the range and remove the keys one by one. The problem is that many index nodes will be shadowed causing loss of sharing with older snapshots and the allocation of many disk-pages.

The algorithm includes two phases. The first phase deallocates keys and nodes in the range. This may, very likely, invalidate the b-tree invariants. The second phase reshapes the tree in a single downward traversal restoring the invariants. This algorithm is based on the delete

algorithm from the Exodus system [14]. The bound on the number of modified pages is $O(\text{tree depth})$.

The removal phase traverses the part of the tree in the range $ofs - end$ and removes all the key-data pairs. If all the children of index node I have been removed then I is also removed. The result is the removal of a complete sub-tree while leaving damaged nodes at the edges. Figure 13 shows an example where the range 11 – 31 is removed from a tree of 11 nodes. A subtree containing keys {15, 16, 20, 25} is completely removed and leaf nodes [10, 11] and [31, 32] underflow (they have less than b entries).

The set of nodes on the edge of the removed-area is called the *cut*. In Figure 13 the cut includes nodes [1, 30], [1, 5, 10], [31, 37], [10, -], and [-, 32]. Nodes in the cut have a potential underflow. Nodes outside the cut have not been touched and they are therefore legal. A node in the cut can have zero, one, or two children that are also in the cut.

A node is *in-danger* if it is in the cut and it has an underflow or, if it may underflow as a result of its children being merged. For example, in Figure 13 node $N = [31, 37]$ is in-danger because if its child [-, 32] is merged with its sibling [37, 40] then N will underflow.

To make it easy to decide whether a node is in-danger the criteria is relaxed. Node N is said to be in-danger if it is in the cut and either (1) it has an underflow (2) or, it has b entries and one child in the cut (3) or, it has $b + 1$ entries and two children in the cut.

The restoration pass traverses the cut in a top-down fashion. It fixes each in-danger node by merging it with or moving entries from neighboring nodes. After fixing, an in-danger node has enough entries so that even if it loses all its in-cut children it will not underflow. In other words, it is made *safe*. This allows the restoration pass to never come back up. An example is shown in Figure 14. The tree from Figure 13 is restored by making a top-down pass on the cut and fixing all the in-danger nodes in it. Two steps are shown, in the first step nodes [1, 5, 10] and [31, 37] are merged. As a side effect, the old root is deleted and the height of the tree is reduced from three to two. In the second step nodes [10, -] and [-, 32] are merged.

Once all in-danger nodes have been fixed the tree contains only legal nodes. Each node-fix step maintains the invariant that all leaves have the same distance to the root. Therefore, at the end of the restoration phase the tree is valid.

Fixing all in-danger nodes can be an overkill, some of the work may be unnecessary. However, the cut contains, in the worst case, two separate paths from the root to the leaves. This is bounded by $2\log_b(N)$. Furthermore, the probability of a node being in-danger is fairly

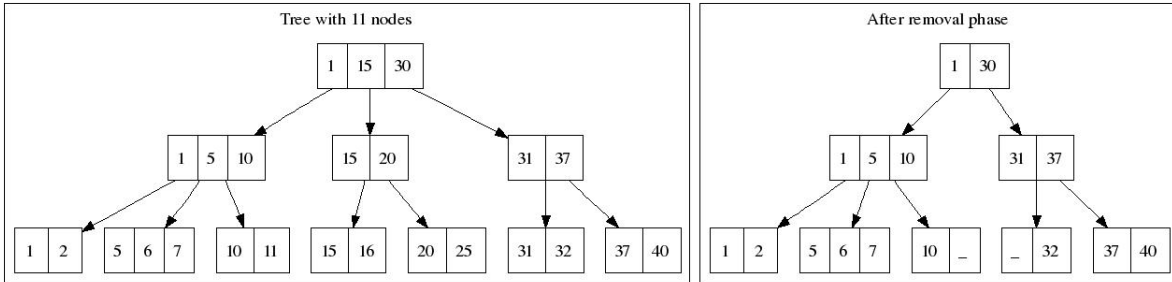


Figure 13: A tree that undergoes a remove-range 11-31 operation.

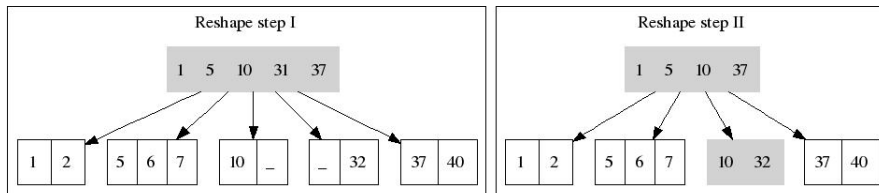


Figure 14: Restoring the tree after the remove phase.

low. By doing extra preparatory work on the way down the algorithm never has to go back up; this simplifies the algorithm considerably.

Without having between b and $2b + 1$ entries in a node the restoration pass does not work. The pass relies on being able to make an in-danger node safe by moving keys into it. For example, examine Figure 15 and assume that the legal bounds are b and $2b - 1$; like in a standard b -tree. Node N is on the edge, it has b entries, and is in-danger. Node N has two neighbors, L and R , that also have b keys. To make N safe 2 keys need to be moved into it. Node N cannot be merged with L because merging them will create a node with $2b$ entries which is illegal. For the same reason N cannot be merged with R . Moreover, keys cannot be shuffled from L or R into node N because the number of keys in L (or R) would be reduced to below b .

5.8 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 $2 \times \text{tree-depth}$. 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

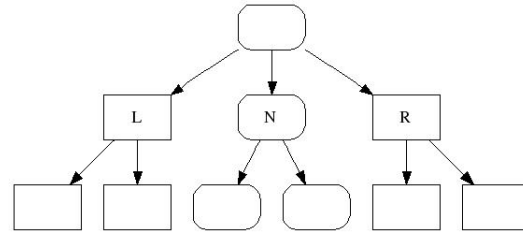


Figure 15: Why $(b, 2b-1)$ does not work. Nodes on the edge have rounded corners.

huge objects and object-catalogs.

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

For remove-range $\log_b(N)$ memory-pages are needed for the remove-phase and a maximum of three pages are needed for the restoration phase. The amount of disk-pages modified is bounded by $4 \times \text{tree-depth}$. This is because the worst case that can occur is that two separate paths in the tree, starting at the root, need to be fixed. In order to fix a node its neighbor must be modified. This means that four nodes need to be touched at each level of the tree.

5.9 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 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 bitmap. 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.

7 Performance

The OSD was built to be part of a storage-controller. It was specified to be able to manage Terra-bytes of disk space with Giga-bytes 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		1283
	write		421
	R+W		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 $\{0, 2, 4, \dots\}$ were chosen as keys; they were inserted sequentially into the tree.

Two base-trees were used T_{235} and T_{150} . 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 check-pointing it. Read-only workloads are run directly on the base tree. For all other workloads, a special procedure is used. A new generation is initiated. The new generation is aged by performing 1000 random insert-key/remove-key operations. Then, the actual workload is applied. At the end, a special debugging-API is used to throw away the new generation and go back to the base tree. 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 number of cache-pages is fixed at initialization time to be five percent of the total number of tree nodes. With 5% as the ratio, a good caching strategy can keep all the index nodes for T_{235} and T_{150} in memory. This means that operations like lookup/remove/insert-key should access, in most cases, one disk page. This looked like an important operating point and was therefore chosen for the experiments.

CPU utilization throughout all the tests was about 1%, the tests were all IO bound.

7.1 Latency

There are six operations whose latency was measured: lookup-key, insert-key, remove-key, append-key, lookup-range, and remove-range. In order to measure latency of operation x an experiment was performed where x was executed 12000 times, and total elapsed time was measured. The latency per operation was computed as the average. Single-key operations (insert, remove, append, lookup) were performed with randomly chosen keys; range operations used a range of 100.

Table 2 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 20ms. The performance of remove-key is about the same as an insert-key; the algo-

rithms are very similar. Append always costs 20us because the pages it operates on are always cached.

Table 3 shows the latency of the range-operations. A range of 100 keys is spread across, roughly, two leaf nodes. Therefore, looking up a range of 100 keys requires, on average, two disk reads. A remove-range for the same range requires not only reading the leaves but also modifying them and the path from the root. This makes it a very costly operation.

As a rule, the costs for T_{150} are higher than T_{235} because less of the tree fits in memory.

Tree	Lookup	Insert	Remove-key	Append
T_{235}	4.657	22.080	22.046	0.016
T_{150}	4.923	23.728	23.711	0.016

Table 2: Latency for single-key operations in milliseconds.

Tree	Lookup-range	Remove-Range
T_{235}	8.686	37.052
T_{150}	11.761	41.274

Table 3: Latency for range operations in milliseconds.

7.2 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 12000 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.

All the tasks were running on a single CPU, due to the use of co-routines. Therefore, throughput gains could only occur as a result of disk parallelism. The disk used in the experimental setup could do 1283 4KB reads per second and 311 4KB read+write per second. This was six times more requests than could be performed by a single synchronous thread. The goal is to make good use of the disk and achieve disk parallelism.

Table 5 shows the results for a single task and for ten tasks. The throughput gain in all cases is x6 or slightly better.

In the *Search-100* workload each lookup translates into a disk-read for the leaf node. This means that the maximal achievable throughput is 1231 requests per second. This is reached for T_{150} and is exceeded for T_{235} . The reason it is exceeded is that T_{235} has less index nodes than T_{150} and some of the memory is used to cache leaves; this improves performance in a small percent of the cases.

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 throughput is bounded by 311 operations. This is achieved or exceeded for both T_{235} and T_{150} .

The *Search-80* and *Modify* workloads are somewhere in between *Search-100* and *Insert*. They perform more than a single disk-read and less than a disk-read + disk-write on average per operation.

Tree	#tasks	Src-100	Src-80	Modify	Insert
T_{235}	10	1286	724	415	367
	1	206	100	51	44
T_{150}	10	1150	617	354	311
	1	173	80	42	36

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

7.3 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.

For example, when the in-memory percent is 5% and all index nodes are in memory each lookup/insert/remove-key operation requires a single free page. If all memory-resident pages are dirty then each operation has to evict a page to disk before reusing it. This adds an additional disk-write to the average operation cost.

In order to assess performance the throughput tests were run against T_{235} . After 20% of the workload was complete, that is, after 2400 operations, a checkpoint was initiated. The checkpoint-code uses an additional task to destage all dirty-pages belonging to the checkpoint.

For the *Search-100* workload there was virtually no degradation. This is because there are no dirty-pages to destage. Other workloads suffer about 10% degradation in performance when ten tasks are used. Somewhat counter-intuitively, when only a single task is used performance is better than without checkpoints. This is because the additional checkpoint task increases parallelism and creates free-pages by destaging dirty pages to disk.

Tree	#tasks	Src-100	Src-80	Modify	Insert
T_{235}	10	1279	636	376	336
	1	205	107	57	50

Table 6: Throughput results, when a checkpoint is performed during the workload.

7.4 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 7. 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 7: 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.

7.5 Different memory percentages

It is interesting to look at performance when the in-memory percentage varies. To this end, several additional measurements were taken. The *Search-100* was run against the T_{235} with the in-memory percentage equal to 100%, 50%, 10%, 5% and 2%. Table 8 summarizes the results.

Tree	% in-memory	1 task	10 tasks
T_{235}	100	88626	88759
	50	409	2460
	10	230	1408
	5	214	1296
	2	174	1065

Table 8: Throughput results, measured in operations per second for T_{235} and the *Search-100* workload.

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: 50%, 10%, and 2% a speedup of about x6 is achieved.

8 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. Douglass. 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. Sedgwick. A Dichromatic Framework for Balanced Trees. In *Nineteenth Annual Symposium on Foundations of Computer Science*, 1978.
- [14] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Storage management for objects in Exodus. In *Object-Oriented Concepts, Databases, and Applications*, 1989.
- [15] M. McKusick, W. Joy, S. Leffler, and R. Fabry. A Fast File System for Unix. *ACM Transactions on Computer Systems*, 1984.
- [16] 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.
- [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.