Stratified B-trees and versioning dictionaries.

Andy Twigg*, Andrew Byde*, Grzegorz Miłoś*, Tim Moreton*, John Wilkes†* and Tom Wilkie*
*Acunu, †Google
firstname@acunu.com

Abstract

External-memory versioned dictionaries are fundamental to file systems, databases and many other algorithms. The ubiquitous data structure is the copy-on-write (CoW) B-tree. Unfortunately, it doesn't inherit the B-tree's optimality properties; it has poor space utilization, cannot offer fast updates, and relies on random IO to scale. We describe the 'stratified B-tree', which is the first versioned dictionary offering fast updates and an optimal tradeoff between space, query and update costs.

1 Introduction

The (external-memory) dictionary is at the heart of any file system or database, and many other algorithms. A dictionary stores a mapping from keys to values. A *versioned dictionary* is a dictionary with an associated version tree, supporting the following operations:

- `update(k,v,x)`: associate value `x` to key `k` in leaf version `v`;
- `range_query(k1,k2,v)`: return all keys (and values) in range [`k1`,`k2`] in version `v`;
- `clone(v)`: return a new child of version `v` that inherits all its keys and values.

Note that only leaf versions can be modified. If `clone` only works on leaf versions, we say the structure is partially-versioned; otherwise it is fully-versioned.

2 Related work

The B-tree was presented in 1972 [?], and it survives because it has many desirable properties; in particular, it uses optimal space, and offers point queries in optimal $O(\log_B N)$ IOs[1]. More details can be found in [?].

A versioned B-tree is of great interest to storage and file systems. In 1986, Driscoll et al. [?] presented the 'path-copying' technique to make pointer-based internal-memory data structures fully-versioned (fully-persistent). Applying this technique to the B-tree gives the copy-on-write (CoW) B-tree, first deployed in EpisodeFS in 1992 [?]. Since then, it has become ubiquitous in file systems and databases, e.g. WAFL [?], ZFS [?], Btrfs [?], and many more.

The CoW B-tree does not share the same optimality properties as the B-tree. Every update requires random IOs to walk down the tree and then to write out a new path, copying previous blocks. Many systems use a CoW B-tree with a log file system, in an attempt to make the writes sequential. Although this succeeds for light workloads, in general it leads to large space blowups, inefficient caching, and poor performance.

3 This paper

For unversioned dictionaries, it is known that sacrificing point lookup cost from $O(\log_B N)$ to $O(\log N)$ allows update cost to be improved from $O(\log_B N)$ to $O((\log N)/B)$. In practice, this is about 2 orders of magnitude improvement for around 3x slower point queries [?]. This paper presents a recent construction, the *Stratified B-tree*, which offers an analogous query/update tradeoff for fully-versioned data. It offers fully-versioned updates around 2 orders of magnitude faster than the CoW B-tree, and performs around one order of magnitude faster for range queries, thanks to heavy use of sequential IO. In addition, it is cache-oblivious [?] and can be implemented without locking. This means it can take advantage of many-core architectures and SSDs.

The downside is that point queries are slightly slower, around 3x in our implementation. However, many applications, particularly for analytics and so-called 'big data' problems, require high ingest rates and range queries, rather than point queries. For these applications, we believe the stratified B-tree is a better choice than the CoW B-tree, and all other known versioned dictionaries. Acunu is developing a commercial open-source implementation of stratified B-trees in the Linux kernel.

[1] We use the standard notation B to denote the block size, and N the total number of elements inserted. For the analysis, we assume entries (including pointers) are of equal size, so B is the number of entries per block.

Structure	**Versioning**	**Update**	**Range query (size Z)**	Space
B-tree [**?**]	None	$O(\log_B N)$ random	$O(\log_B N + Z/B)$ random	$O(N)$
CoW B-tree [ZFS]	Full	$O(\log_B N_v)$ random	$O(\log_B N_v + Z/B)$ random	$O(NB \log_B N)$
MVBT [**?**]	Partial	$O(\log_B N_v)$ random	$O(\log_B N_v + Z/B)$ random	$O(N + V)$
Lanka et al. (fat field) [**?**]	Full	$O(\log_B N_v)$ random	$O((\log_B N_v)(1 + Z/B))$ random	$O(N + V)$
Stratified B-tree [this paper]	Full	$O^*((\log N_v)/B)$ sequential	$O^*(\log N_v + Z/B)$ sequential	$O(N + V)$

Table 1: Comparing the cost of basic operations on versioned dictionaries. Bounds marked $O^*(\cdot)$ are amortized over operations on a given version.

Table **??** summarises our results and known data structures. We use N for the total number of keys written over all versions, and N_v for the number of keys *live* (accessible) in version v, and V for the total number of versions (see Preliminaries in Section **??**).

3.1 CoW B-tree

The CoW B-tree is the classic versioned dictionary in file systems, storage and other external-memory algorithms. The basic idea is to have a B-tree with many roots, one for each version. Nodes are versioned and updates can only be done to a node of the same version as the update. To perform an update in $v2$, one ensures there is a suitable root for $v2$ (by duplicating the root for $v1$, the parent of $v2$, if necessary), then follows pointers as in a normal B-tree; every time a node is encountered with version other than $v2$, it is copied to make a new node with version $v2$, and the parent's pointer updated before the update continues down the tree. A lookup proceeds as in a B-tree, starting from the appropriate root. More details can be found in [**?**].

It has three major problems: **slow updates**, since each update may cause a new path to be written – updating a 16-byte key in a tree of depth 3 with 256K block size requires 3x256K random reads and writing 768K of data. Since these nodes cannot be garbage collected unless a version is deleted, they represent a large **space blowup**. Finally, it **relies on random IO**, for both updates and range queries. Over time, the leaves tend to be scattered randomly, which causes problems for range queries, garbage collection and prefetching.

4 Multiversion B-trees

The multi-version B-tree (MVBT) [**?**] offers the same query/update bounds as the CoW B-tree, but with asymptotically optimal $O(N)$ space. However, it is only partially-versioned. The basic idea is to use *versioned pointers* inside nodes – each pointer stores the range of (totally-ordered) versions for which the pointer is *live*. A query for version v starts by finding the root node for v, then at every node, extracting the set of pointers live at v and treating these as an unversioned B-tree node. Updates are more complex and require an additional 'version split' operation in addition to the standard key split.

Soules et al. [**?**] compare the metadata efficiency of a versioning file system using both CoW B-trees and a structure (CVFS) based on the MVBT. They find that, in many cases, the size of the CoW metadata index exceeds the dataset size. In one trace, the versioned data occupies 123GB, yet the CoW metadata requires 152GB while the CVFS metadata requires 4GB, a saving of 97%.

5 Stratified B-trees

Structure. The high-level structure is vaguely similar to the logarithmic method (e.g. employed by the COLA of Bender et al. [**?**]). We store a collection of arrays of sorted `(key, version, value)` elements, arranged into levels, with 'forward pointers' between arrays to facilitate searches. Each element is written (k, v, x), where k is a key, v is a version id, and x is either a value or pointer to a value. Each array A is tagged with some subtree W of the version tree, so we write (A, W). Arrays in the same level have disjoint version sets, hence *stratified* in version space. Arrays in level l have size roughly 2^{l+1}.

Preliminaries. Elements within an array (A, W) are ordered lexicographically by (k, v). We assume keys come equipped with a natural ordering (e.g., byte-order), and versions are ordered by decreasing DFS (depth-first-search) number in the version tree. This has the property that for any v, all descendants of v appear in a contiguous region to its left. For a version v, we write $W[v]$ for the subtree of W descendant from v. We also write, for simplicity, V to represent both the global version tree and the number of versions in it (its use will be clear from the context).

An element (k, v, x) is said to *live* at version w iff v is a descendant of w, and k has not been rewritten between v and w. An element (k, v, x) is *lead* at version v. For an array (A, W), we write $live(A, W)$ to count all the elements live at some $w \in W$, and similarly for $lead(A, W)$. We use N to count the total number of elements inserted, and N_v to count the number of elements live at version v.

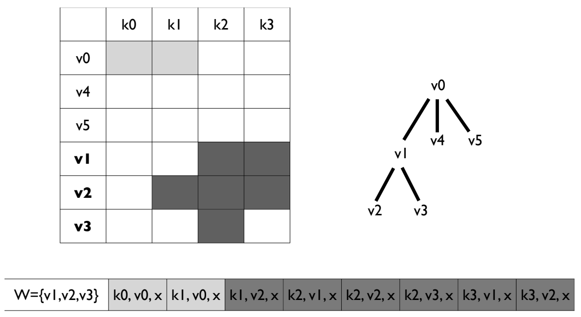

Figure 1: A versioned array, a version tree and its layout on disk. Versions $v1, v2, v3$ are tagged, so dark entries are lead entries. The entry (k_0, v_0, x) is written in v_0, so it is not a lead entry, but it is live at v_1, v_2 and v_3. Similarly, (k_1, v_0, x) is live at v_1 and v_3 (since it was not overwritten at v_1) but not at v_2. The live counts are as follows: $live(v_1) = 4, live(v_2) = 4, live(v_3) = 4$, and the array has density $\frac{4}{8}$. In practice, the on-disk layout can be compressed by writing the key once for all the versions, and other well-known techniques.

We say an array (A, W) has *density* δ if, for all $w \in W$, at least a fraction δ of the elements in A are *live* at w. We call an array *dense* if it has density $\geq \frac{1}{6}$. The importance of density is the following: if density is too low, then scanning an array to answer a range query at version v will involve skipping over many elements not live at v. On the other hand, if we insist on density being too high, then many elements must be duplicated (in the limit, each array will simply contain all the live elements for a single version), leading to a large space blowup. The central part of our construction is a merging and splitting process that guarantees that every array has both density and lead fraction $\Omega(1)$. This is enough to guarantee only a constant $O(1)$ space and query blowup.

The notion of a *split* is important for the purpose of breaking large arrays into smaller ones. For an array (A, W) and version set $X \subseteq W$, we say $\text{split}((A, W), X) \subseteq A$ contains all the elements in A live at any $x \in X$.

Notation in figures. Figures **??** and **??** give examples of arrays and various split procedures. We draw a versioned array as a matrix of (key,version) pairs, where the shaded entries are present in the array. For each array (A, W), versions in W are written in bold font (note that arrays may have entries in versions not in W – these entries are duplicates, they are live in some version in W, but are not lead). Figure **??** gives an example of a versioned array.

Queries. In the simplest case, a point query can be answered by considering each level independently. Unlike the COLA, each level may have many arrays. We guarantee that a query for version v examines at most one array per level: examine the array (A, W) where W contains the closest ancestor to v. In the simplest construction, we maintain a per-array B-tree index for the keys in that array, and a Bloom filter on its key set. A query at v involves querying the Bloom filter for each array selected as above, then examining the B-tree in those arrays matching the filter. If multiple results are found, return the one written in the nearest ancestor to v – or in the lowest level if there is more than one such entry. In practice, the internal nodes of each B-tree can be held in memory so that this involves $O(1)$ IOs per array.

For a range query, we find the starting point in each array, and merge together the results of iterating forwards in each array. The density property implies that, for large range queries, simply scanning the arrays with the appropriate version tags will use an asymptotically optimal number of IOs. For small range queries, we resort to a more involved 'local density' argument to show that, over a large enough number of range queries, the average range query cost will also be optimal. Indeed, doing better with $O(N)$ space is known to be impossible. More details can be found in [**?**].

Insertions, promotions and merging. An insert of (k, v, x) consists of promoting the singleton array $(\{k, v, x\}, \{v\})$ into level 0. In general, an array $\mathcal{S} = (A, W)$ promoted to level l always has a unique root w in W, so we merge it with the unique array (A', W') at level l that is tagged with the closest ancestor to v (if such an array exists) to give an array $(\mathcal{A}, \mathcal{W}) = (A \cup A', W \cup W')$. If no such array exists, we leave $\mathcal{S}$ in place.

After merging, $(\mathcal{A}, \mathcal{W})$ may be too large to exist at level l. We employ two procedures to remedy this. First, we call **find-promotable** in order to extract any possible subarrays that are large and dense enough to survive at level $l + 1$. Secondly, on the remainder (which may be the entire array if no promotable subarray was found), we apply **density amplification**, which produces a collection of dense subarrays that all have the required size.

Find-promotable. This proceeds by searching for the highest (closest to root) version $v \in \mathcal{W}$ such that $X = \text{split}(A, \mathcal{W}[v])$ has size $\geq 2^{l+1}$ (thus being too large for level l) and has a sufficiently high live and lead fraction. If such a version is to be found then it is extracted and promoted to level $l + 1$, leaving the remainder $(\mathcal{A}', \mathcal{W}') = \text{split}(\mathcal{A}, \mathcal{W} \setminus \mathcal{W}[v])$. Figure **??** gives an example of find-promotable.

Density amplification. Now we consider the remainder after calling find-promotable. There may be a version $w \in \mathcal{W}'$ that was originally dense but is no longer dense in the larger $(\mathcal{A}', \mathcal{W}')$. To restore density we subdivide the array into several smaller arrays (A'_i, W'_i), each of which is dense. Crucially, by carefully choosing which sets of versions to include in which array, we are able

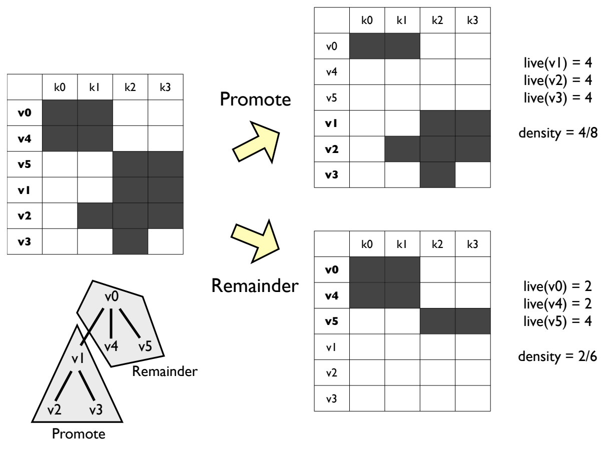

Figure 2: Example of find-promotable. The merged array is too large to exist at the current level (it has size 12, whereas level $l = 3$ requires size < 8, say). We promote to the next level the subtree under version $v1$, which gives the promoted array of size 8 and density $\frac{4}{8} = \frac{1}{2}$. Note that the entries $(k_0, v_0), (k_1, v_0)$ are duplicated, since they are also live at v_1, v_2, v_3. The remainder array has size 6, so it can remain at the original level.

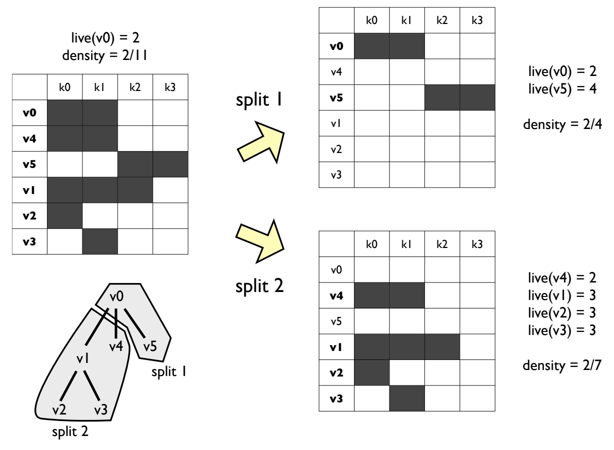

Figure 3: Example of density amplification. The merged array has density $\frac{2}{11} < \frac{1}{6}$, so it is not dense. We find a split into two parts: the first split $(A_1, \{v_0, v_5\})$ has size 4 and density $\frac{1}{2}$. The second split $(A_2, \{v_4, v_1, v_2, v_3\})$ has size 7 and density $\frac{2}{7}$. Both splits have size < 8 and density $\geq \frac{1}{6}$, so they can remain at the current level.

to bound the amount of work done during density amplification, and to guarantee that not too many additional duplicate elements are created.

We start at the root version and greedily search for a version v and some subset of its children whose split arrays can be merged into one dense array at level l. More precisely, letting $\mathcal{U} = \bigcup_i \mathcal{W}'[v_i]$, we search for a subset of v's children $\{v_i\}$ such that

$$|\text{split}(\mathcal{A}', \mathcal{U})| < 2^{l+1}.$$

If no such set exists at v, we recurse into the child v_i maximizing $|\text{split}(\mathcal{A}', \mathcal{W}'[v_i])|$. It is possible to show that this always finds a dense split. Once such a set $\mathcal{U}$ is identified, the corresponding array is written out, and we recurse on the remainder $\text{split}(\mathcal{A}', \mathcal{W}' \setminus \mathcal{U})$. Figure **??** gives an example of density amplification.

6 Practicalities

Consistency. We would like the data structure to be *consistent* – at any time, the on-disk representation is in a well-defined and valid state. In particular, we'd like it to always be the case that, after a crash or power loss, the system can continue without any additional work (or very little) to recover from the crash. Both CoW and Stratified B-tree implement consistency in a similar way: the difference between the new and old data structure is assembled in such a way that it can be abandoned at any time up until a small in-memory change is made at the end. In the case of CoW it is the root for version v: until the table mapping v to root node is updated, the newly copied nodes are not visible in searches, and can be recycled in the event of a crash. Likewise for the Stratified B-tree arrays are not destroyed until they are no longer in the search path, thus ensuring consistency.

Concurrency. With many-core architectures, exploiting concurrency is crucial. Implementing a good concurrent B-tree is difficult. The Stratified B-tree can naturally be implemented with minimal locking. Each per-array B-tree is built bottom-up (a node is only written once all its children are written), and once a node is written, it is immutable. If the B-tree T is being constructed as a merge of T_1, T_2, then there is a *partition key* k such that all nodes with key less than k are immutable; based on k, queries can be sent either to T, or to T_1 and T_2. Each merge can thus be handled with a single writer without locks. In the Stratified B-tree, there are many merges ongoing concurrently; each has a separate writer.

Allocation and low free space. When a new array of size k needs to be written, we try to allocate it in the first free region of size $\geq k$. If this fails, we use a 'chunking strategy': we divide each array into contiguous *chunks* of size $c >> B$ (currently 10MB). During a merge, we read chunks from each input array until we have a chunk formed in memory to output. When a chunk is no longer needed from the input chunk arrays, it can be deallocated and the output chunk written there. This doesn't guarantee that the the entire array is sequential on disk, but it is sequential to within the chunk size, which is sufficient in practice to extract good performance, and only

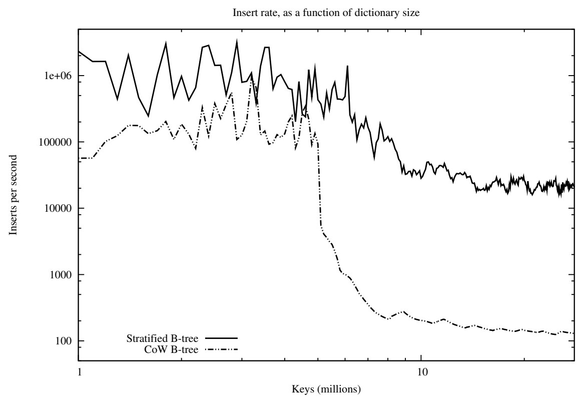

Figure 4: Insert performance with 1000 versions.

uses $O(c)$ extra space during merges - thus the system degrades gracefully under low free space conditions.

7 Experimental results

We implemented prototypes of the Stratified B-tree and CoW B-tree (using in-place updates) in OCaml. The machine had 1GB memory available, a 2GHz Athlon 64 processor (although our implementation was only single-threaded) and a 500GB SATA disk. We used a block size of 32KB; the disk can perform about 100 such IOs/s. We started with a single root version and inserted random 100 byte key-value pairs to random leaf versions, and periodically performed range queries of size 1000 at a random version. Every 100,000 insertions, we create a new version as follows: with probability 1/3 we clone a random leaf version and w.p. 2/3 we clone a random internal node of the version tree. The aim is to keep to the version tree 'balanced' in the sense that there are roughly twice as many internal nodes as leaves.

Figures **??** and **??** show update and range query performance results for the CoW B-tree and the Stratified B-tree. The B-tree performance degrades rapidly when the index exceeds internal memory available. The right plot shows range query performance (elements/s extracted using range queries of size 1000). The Stratified B-tree beats the CoW B-tree by a factor of more than 10. The CoW B-tree is limited by random IO here[2], but the Stratified B-tree is CPU-bound (OCaml is single-threaded). Preliminary performance results from a highly-concurrent in-kernel implementation suggest that well over 500k updates/s are possible with 16 cores.

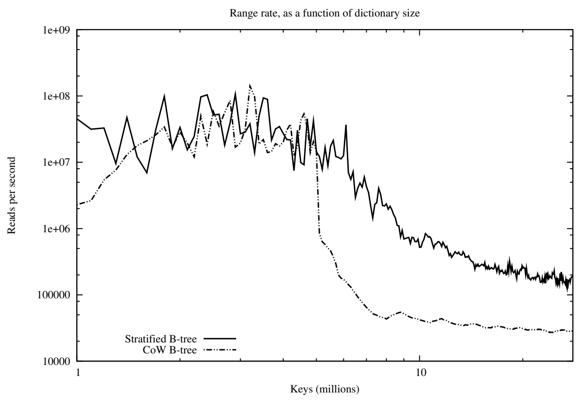

Figure 5: Range query performance with 1000 versions.

[2](100/s*32KB)/(200 bytes/key) = 16384 key/s