Confluently Persistent Tries for Efficient Version Control

Erik D. Demaine[1*], Stefan Langerman[2**], and Eric Price[1]

[1] MIT Computer Science and Artificial Intelligence Laboratory,
32 Vassar Street, Cambridge, MA 02139, USA, {edemaine,ecprice}@mit.edu
[2] Computer Science Department, Université Libre de Bruxelles,
CP 212, Bvd. du Triomphe, 1050 Brussels, Belgium, stefan.langerman@ulb.ac.be

Abstract. We consider a data-structural problem motivated by version control of a hierarchical directory structure in a system like Subversion. The model is that directories and files can be moved and copied between two arbitrary versions in addition to being added or removed in an arbitrary version. Equivalently, we wish to maintain a confluently persistent trie (where internal nodes represent directories, leaves represent files, and edge labels represent path names), subject to copying a subtree between two arbitrary versions, adding a new child to an existing node, and deleting an existing subtree in an arbitrary version.

Our first data structure represents an n-node degree-Δ trie with $O(1)$ "fingers" in each version while supporting finger movement (navigation) and modifications near the fingers (including subtree copy) in $O(\lg \Delta)$ time and space per operation. This data structure is essentially a locality-sensitive version of the standard practice—path copying—costing $O(d \lg \Delta)$ time and space for modification of a node at depth d, which is expensive when performing many deep but nearby updates. Our second data structure supporting finger movement in $O(\lg \Delta)$ time and no space, while modifications take $O(\lg n)$ time and space. This data structure is substantially faster for deep updates, i.e., unbalanced tries. Both of these data structures are functional, which is a stronger property than confluent persistence. Without this stronger property, we show how both data structures can be sped up to support movement in $O(\lg \lg \Delta)$, which is essentially optimal. Along the way, we present a general technique for global rebuilding of fully persistent data structures, which is nontrivial because amortization and persistence do not usually mix. In particular, this technique improves the best previous result for fully persistent arrays and obtains the first efficient fully persistent hash table.

1 Introduction

This paper is about a problem in persistent data structures motivated by an application in version control. We begin by describing the motivating application, then our model of the underlying theoretical problem, followed by our results.

* Supported in part by MADALGO — Center for Massive Data Algorithmics — a Center of the Danish National Research Foundation.
** Chercheur qualifié du F.R.S.-FNRS.

Version control. Increasingly many creative works on a computer are stored in a *version control system* that maintains the history of all past versions. The many motivations for such systems include the ability to undo any past mistake (in the simplest form, never losing a file), and the ability for one or many people to work on multiple parallel branches (versions) that can later be merged. Source code has been the driving force behind such systems, ranging from old centralized systems like RCS and CVS, to the increasingly popular centralized system Subversion, to recent distributed systems like Bazaar, darcs, GNU arch, Git, Monotone, and Mercurial. By now version control is nearly ubiquitous for source code and its supporting documentation. We also observe a rise in the use of the same systems in academic research for books, papers, figures, classes, etc.[3] In more popular computer use, Microsoft Word supports an optional form of version control ("change tracking"), Adobe Creative Suite (Photoshop, Illustrator, etc.) supports optional version control ("Version Cue"), and most Computer Aided Design software supports version control in what they call Product Data Management (e.g., Autodesk Productstream for AutoCAD and PDMWorks for SolidWorks). Entire modern file systems are also increasingly version controlled, either by taking periodic global snapshots (as in AFS, Plan 9, and WAFL), or by continuous change tracking (as in Apple's new Time Machine in HFS+, and in experimental systems CVFS, VersionFS, Wayback, and PersiFS [PCD05]). As repositories get larger, even to the point of entire file systems, high-performance version control is in increasing demand. For example, the Git system was built simply because no other free system could effectively handle the Linux kernel.

Requirements for version control. Most version control systems mimic the structure of a typical file system: a tree hierarchy of directories, each containing any number of linear files. Changes to an individual file can therefore be handled purely locally to that file. Conceptually these changes form a tree of versions of the file, though all systems represent versions implicitly by storing a delta ("diff") relative to the parent. In this paper, we do not consider such file version tracking, because linear files are relatively easy to handle.

The more interesting data structural challenge is to track changes to the hierarchical directory structure. All such systems support addition and removal of files in a directory, and creation and deletion of empty subdirectories. In addition, every system since Subversion's pioneering innovation supports moving or copying an entire subdirectory from one location to another, possibly spanning two different versions. This operation is particularly important for merging different version branches into a common version.

Persistent trie model. Theoretically, we can model version control of a hierarchical directory structure as a confluently persistent trie, which we now define.

A *trie* is a rooted tree with labeled edges. In the version-control application, internal nodes represent directories, leaves represent files, and edge labels represent file or directory names.[4] The natural queries on tries are navigation: placing

[3] For example, this paper is maintained using Subversion.

[4] We assume here that edge labels can be compared in constant time; in practice, this property is achieved by hashing the file and directory name strings.

a finger at the root, moving a finger along the edge with a specified label, and moving a finger from a node to its parent. We assume that there are $O(1)$ fingers in any single version of the trie; in practice, two fingers usually suffice. Each node has some constant amount of information which can be read or written via a finger; for example, each leaf can store a pointer to the corresponding file data structure. The structural changes supported by a trie are insertion and deletion of leaves attached to a finger (corresponding to addition and removal of files), copying the entire subtree rooted at one finger to become a new child subtree of another finger (corresponding to copying subdirectories), and deleting an entire subtree rooted at one finger (enabling moving of subdirectories). Subtree copying propagates any desired subset of the fingers of the old subtree into the new subtree, provided the total number of fingers in the resulting trie remains $O(1)$.

The trie data structure must also be "confluently persistent". In general, persistent data structures preserve old versions of themselves as modifications proceed. A data structure is *partially persistent* if the user can query old versions of the structure, but can modify only the most recent version; in this case, the versions are linearly ordered. A data structure is *fully persistent* if the user can both query and modify past versions, creating new branches in a tree of versions. The strongest form of persistence in *confluent persistence*, which includes full persistence but also supports certain "meld" operations that take multiple versions of the and produce a new version; then the version dependency graph becomes a directed acyclic graph (DAG). The version-control application demands confluent persistence because branch merging requires the ability to copy subdirectories (subtrees) from one version into another.

Related work in persistence. Partial and full persistence were mostly solved in the 1980's. In 1986, Driscoll et al. [DSST89] developed a technique that converts any pointer-based data structure with bounded in-degree into an equivalent fully persistent data structure with only constant-factor overhead in time and space for every operation. In 1989, Dietz [Die89] developed a fully persistent array supporting random access in $O(\lg \lg m)$ time, where m is the number of updates made to any version. This data structure enables simulation of an arbitrary RAM data structure with a log-logarithmic slowdown. Furthermore, this slowdown is essentially optimal, because fully persistent arrays have a lower bound of $\Omega(\lg \lg n)$ time per operation in the powerful cell-probe model.[5]

More relevant is the work on confluent persistence. The idea was first posed as an open problem by [DSST89]. In 1994, Driscoll et al. [DST94] defined confluence and gave a specific data structure for confluently persistent catenable lists. In 2003, Fiat and Kaplan [FK03] developed the first and only general methods for making a pointer-based data structure confluently persistent, but the slowdown is often suboptimal. In particular, their best deterministic result has a linear worst-case slowdown. Although their randomized result has a polylogarithmic amortized slowdown, it can take a linear factor more space per modification,

[5] Personal communication with Mihai Pătraşcu, 2008. The proof is based on the predecessor lower bounds of [PT07].

and furthermore the answers are correct only with high probability; they do not have enough time to check the correctness of their random choices.

Fiat and Kaplan [FK03] also prove a lower bound on confluent persistence. They essentially prove that most interesting confluently persistent data structures require $\Omega(\lg p)$ space per operation in the worst case, even with randomization, where p is the number of paths in the version DAG from the root version to the current version. Note that p can be exponential in the number m of versions, as in Figure 1, resulting in a lower bound of $\Omega(m)$ space per operation. The lower bound follows from the possibility of having around p addressable nodes in the data structure; in particular, it is easy to build an exponential amount of data (albeit with significant repetition) using a linear number of confluent operations. However, their $\Omega(\lg p)$ lower bound requires a crucial and unreasonable assumption: that all nodes of the structure can be addressed at any time. From the perspective of actually using a data structure, it is much more natural for the user to have to locate the data of interest using a sequence of queries. For this reason, our use of trie traversals by a constant number of fingers is both natural and critical to our success.

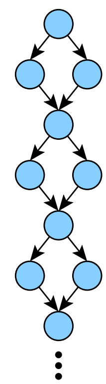

Fig. 1. This version DAG has exponentially many paths from top to bottom, and can result in a structure with an exponential data.

Functional data structures. Given the current lack of general transformations into confluently persistent data structures, efficient such structures seem to require exploiting the specific problem. One way to attain confluent persistence is to design a *functional* data structure, that is, a read-only (pointer-based) data structure. Such a data structure can create new cells with new initial contents, but cannot modify the contents of old cells. Each modifying operation requires a pointer to the new version of the data structure, described by a newly created cell. Functional data structures have many useful properties other than confluent persistence; for example, multiple threads can use functional data structures without locking. Pippenger [Pip97] proved a logarithmic-factor separation between the best pointer-based data structure and the best functional data structure for some problems. On the other hand, many common data structures can be implemented functionally with only a constant-factor overhead; see Okasaki [Oka98]. One example we use frequently is a functional *catenable deque*, supporting insertion and deletion at either end and concatenation of two deques in constant time per operation [Oka98].

Path copying. Perhaps the simplest technique for designing functional data structures is *path copying* [Oka98]. This approach applies to any tree data structure where each node modification depends on only the node's subtree. Whenever we would modify a node v in the ephemeral (nonpersistent) structure, we instead create new copies of v and all ancestors of v. Because nodes depend on only their subtrees and the data structure becomes functional (read only), we can safely reuse all other nodes. Figure 2 shows an example of path copying in a binary search tree (which achieves logarithmic worst-case performance).

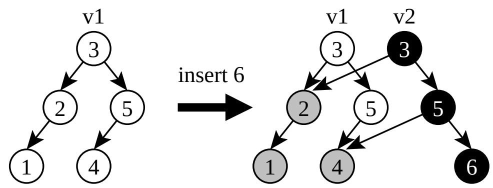

Fig. 2. Path copying in a binary search tree. The old version (v1) consists of white and grey nodes; the new version (v2) consists of black and grey nodes.

Version control systems including Subversion effectively implement path copying. As a result, modifications to the tree have a factor-$\Theta(d)$ overhead, where d is the depth of the modified node. More precisely, for a pointer-based data structure, we must split each node of degree Δ into a binary tree of height $O(\lg \Delta)$, costing $O(d \lg \Delta)$ time and space per update.

Imbalance. The $O(d \lg \Delta)$ cost of path copying is potentially very large because the trie may be extremely unbalanced. For example, if the trie is a path of length n, and we repeatedly insert n leaves at the bottommost node, then path copying requires $\Omega(n^2)$ time and space.

Douceur and Bolosky [DB99] studied over 10,000 file systems from nearly 5,000 Windows PCs in a commercial environment, totaling 140 million files and 10.5 terabytes. They found that d roughly follows a Poisson distribution, with 15% of all directories having depth at least eight. Mitzenmacher [Mit03] studies a variety of theoretical models for file-system creation which all imply that d is usually logarithmic in n.

Our results. We develop four trie data structures, two of which are functional and two of which are efficient but only confluently persistent; see Table 1. All four structures effectively break through the lower bound of Fiat and Kaplan.

Our first functional trie enables exploiting locality of reference among any constant number of fingers. Both finger movement (navigation) and modifications around the fingers (including subtree copy) cost $O(\lg \Delta)$ time and space per operation, where Δ is the average degree of the nodes directly involved. Note that navigation operations require space as well, though the space is permanent only if the navigation leads to a modification; stated differently, the space cost of a modification is effectively $O(t \lg \Delta)$ where t is the distance of the finger from its last modification. This data structure is always at least as efficient as path

Method	Finger movement		Modifications	
	Time	Space	Time	Space
Path copying	$\lg \Delta$	0	d	d
Locality-sensitive (functional)	$\lg \Delta$	$\lg \Delta$	$\lg \Delta$	$\lg \Delta$
Locality-sensitive (fully persistent)	$\lg\lg \Delta$	$\lg\lg \Delta$	$\lg\lg \Delta$	$\lg\lg \Delta$
Globally balanced (functional)	$\lg \Delta$	0	$\lg n$	$\lg n$
Globally balanced (fully persistent)	$\lg\lg \Delta$	0	$\lg n$	$\lg n$

Table 1. Time and space complexity of data structures described in this paper. Operations are on an n-node trie at a node of depth d and degree Δ.

copying, and much more efficient in the case of many deep but nearby modifications. In particular, the quadratic example of inserting n leaves at the bottom of a length-n path now costs only $O(n \lg \Delta)$ time and space.

Our second functional trie guarantees $O(\lg n)$ time and space per modification, with no space required by navigation, while preserving $O(\lg \Delta)$ time per navigation. This data structure is substantially more space-efficient than the first data structure whenever modifications are deep and dispersed. For example, if we insert n leaves alternately at the top and at the bottom of a length-n path, then the time cost from navigation is $\Theta(n^2)$, but the space cost is only $O(n \lg n)$. The only disadvantage is that nearby modifications still cost $\Theta(\lg n)$ time and space, whereas the $O(t \lg \Delta)$ cost of the first data structure can be a bit smaller.

Our two confluently persistent trie data structures are based on the functional data structures, replacing each height-$O(\lg \Delta)$ binary tree representation of a degree-Δ trie node with a new log-logarithmic fully persistent hash table. For the first structure, we obtain an exponentially improved bound of $O(\lg \lg \Delta)$ time and space per operation. For the second structure, we improve the movement cost to $O(\lg \lg \Delta)$ time (and no space). These operations have matching $\Omega(\lg \lg \Delta)$ time lower bounds because in particular they implement fully persistent arrays.

To our knowledge, efficient fully persistent hash tables have not been obtained before. The obvious approach is to use standard hash tables while replacing the table with the fully persistent array of Dietz [Die89]. There are two main problems with this approach. First, the time bound for fully persistent arrays is $O(\lg \lg m)$, where m is the number of updates to the array, but this bound can be substantially larger than even the size Δ of the hash table. Second, hash tables need to dynamically resize, and amortization does not mix well with persistence: the hash table could be put in a state where it is about to pay a huge cost, and then the user modifies that version repeatedly, each time paying the huge cost.

The solution to both problems is given by a new general technique for global rebuilding of a fully persistent data structure. The classic global rebuilding technique from ephemeral data structures, where the data structure rebuilds itself from scratch whenever its size changes by a constant factor, does not apply in the persistent context. Like the second problem above, we cannot charge the linear rebuild cost to the elements that changed the size, because such elements might get charged many times, even with de-amortized global rebuilding. Nonetheless, we show that clever choreography of specific global rebuilds works in the fully persistent context. As a result, we improve fully persistent arrays to support operations in $O(\lg \lg \Delta)$ time and space, where Δ is the current size of the array, matching the $\Omega(\lg \lg \Delta)$ lower bound. We also surmount the amortization and randomization issues with this global rebuilding technique.

2 Locality-Sensitive Functional Data Structure

Our first functional data structure represents a trie T with a set F of $O(1)$ fingers $f_1, f_2, \ldots, f_k$ while supporting finger movements, leaf insertion and deletion, and subtree copies and removals in $O(\lg \Delta)$ time and space per operation.

Let T' be the Steiner tree with terminals f_i, that is, the union of shortest paths between all pairs of fingers. Let PF be the set of nodes with degree at least 3 in T' that are not in F. The elements of PF are called *prosthetic fingers* and will be maintained dynamically. Let $F' = F \cup PF$. Note that $|F'| \leq 2k = O(1)$. Let T'' be the compressed Steiner tree obtained from T' by contracting every vertex not in F' (each of degree 2). For any two fingers in F' that are adjacent in T'', the shortest path in T' connecting them is called a *tendon*. A subtree of T that does not contain any finger of F' is called a *knuckle*.

We represent a tendon by a deque, where each element of the deque corresponds to a vertex of T and is represented by a balanced tree of depth $O(\lg \Delta)$ containing the neighbors of that vertex in T other than those in the deque. Each of the nodes in that tree contains a knuckle. The tendon also contains the labels of the two fingers to which it is attached. We represent a knuckle either by a vertex containing a balanced tree of depth $O(\lg \Delta)$, where each node of the tree represents a neighbor of that vertex in T', or by a deque representing a path starting at the root, whose structure is identical to that of the tendon.

The functional data structure stores all fingers in F' in a balanced binary search tree called the *hand*, where all fingers are ordered by the label of the corresponding node. Every finger stores a balanced tree of depth $O(1)$ for the tendons attached to this finger, and another balanced tree of depth $O(\lg \Delta)$ for the knuckles attached to this finger.

To complete this description, it remains to show how to perform update operations and how to move fingers. When performing an update at a finger, we modify the balanced tree attached to that finger: adding a neighbor for a leaf insertion or subtree copy, and deleting a neighbor for a leaf deletion or subtree removal. Then we use the path-copying technique on both that tree and the hand. To move a finger, we essentially transfer vertices between its neighboring nodes, knuckles, and tendons:

1. If a finger enters a neighboring knuckle (stored in a node of its balanced tree), we will move the finger to its new position. This might involve extracting the new finger from a deque and modifying a constant number of neighbors in its balanced tree. We have two cases:
   (a) If the finger has degree 1 in T'', then it is attached to exactly one tendon τ. We insert into τ the vertex at the previous position of the finger.
   (b) If the finger has degree 2 or more in T'', then after the move that vertex has degree at least 3 and we create a new prosthetic finger at that position. The hand must then be modified so that the tendons that were adjacent to the finger now point to the prosthetic finger. This costs $O(1)$.
2. If a finger moves along a tendon, we proceed similarly, but now, the previous finger is either transferred to a neighboring knuckle or tendon, or becomes a new prosthetic finger. The new vertex for the finger is extracted from the tendon, or if the tendon is empty, two fingers become equal.

The operations change only $O(1)$ nodes from the balanced trees or the deques outside the hand, for a cost of $O(\lg \Delta)$, and modify the hand, which has size $O(1)$.

3 Globally Balanced Functional Data Structure

Our second functional data structure represents the trie as a balanced binary tree, then makes this tree functional via path copying. Specifically, we will use a balanced representation of tries similar to link-cut trees of Sleator and Tarjan [ST83]. This representation is natural because the link and cut operations are essentially subtree copy and delete.[6] Sleator and Tarjan's original formulation of link-cut trees cannot be directly implemented functionally via path copying, and we explain how to address these issues in Section 3.1. In addition to being able to modify the trie, we need to be able to navigate this representation as we would the original trie. We discuss how to navigate in Section 3.2.

A key element of our approach is the *finger*. In addition to the core data structure representing a trie, we also maintain a constant number of locations in the trie called fingers. A finger is a data structure in itself, storing more than just a pointer to a node. Roughly speaking, a finger consists of pointers to all ancestors of that node in the balanced tree, organized to support several operations for computing nearby fingers. These pointers contrast nodes in the balanced tree representation, which do not even store parent pointers. Modifications to our data structure must preserve fingers to point to the same location in the new structure, but fortunately there are only finitely many fingers to maintain. Section 3.4 details the implementation of fingers.

3.1 Functional Link-Cut Trees. In the original link-cut trees [ST83], nodes store pointers to other nodes outside of their subtree, which prevents us from applying path copying. We show how to modify link-cut trees to avoid such pointers and thereby obtain functional link-cut trees via path copying.

There are multiple kinds of link-cut trees; we follow the worst-case logarithmic link-cut trees of [ST83, Section 5]. These link-cut trees decompose the trie into a set of "heavy" paths, and represent each heavy path by a globally biased binary tree [BST85], tied together into one big tree which we call the *representation tree*. An edge is *heavy* if more than half of the descendants of the parent are also descendants of the child. A *heavy path* is a contiguous sequence of heavy edges. Because any node has at most one heavy child, we can decompose the trie into a set of heavy paths, connected by *light* (nonheavy) edges. Following a light edge decreases the number of nodes in the subtree by at least a factor of two, so any root-to-leaf path intersects at most $\lg n$ heavy paths. The *weight* w_v of a node v is 1 plus the number of descendants of v through a light child of v. The depth of a node v in its globally biased search tree T is at most $\lg\left[\left(\sum_{u\in T} w_u\right)/w_v\right]$. If the root-to-leaf path to a node v intersects k heavy paths and w_i is the weight of the last ancestor of v in the ith heavy path down to v, then the total depth of v is $\lceil\lg(n/w_1)\rceil + \lceil\lg(w_1/w_2)\rceil + \cdots + \lceil\lg(w_{k-1}/w_k)\rceil$, or $O(\lg n)$.

Link-cut trees augment each node in a globally biased search tree to store substantial extra information. Most of this information depends only on the subtree of the node, which fits the path-copying framework. The one exception is the parent of the node, which we cannot afford to store. Instead, we require a

[6] Euler-tour trees are simpler, but linking multiple occurrences of a node in the Eulerian tour makes path copying infeasible.

parent operation on fingers, which returns a finger pointing to the parent node in the representation tree. For this operation to suffice, we must always manipulate fingers, not just pointers to nodes. This restriction requires one other change to the augmentation in a link-cut tree. Namely, instead of storing pointers to the minimum (leftmost) and maximum (rightmost) descendants in the subtree, each node stores *relative fingers* to these nodes. Roughly speaking, such a relative finger consists of the nodes along the path from the node to the minimum (maximum). We require the ability to concatenate partial fingers, in this case, the finger of the node with a relative finger to the minimum or maximum, resulting in a (complete) finger to the latter.

To tie together the globally biased search trees for different heavy paths, the link-cut tree stores, for each node, an auxiliary globally biased tree of its light children. More precisely, leaves of the auxiliary tree point to (the root of) the globally biased search tree representing the heavy path starting at such light children. In addition, the top node in a heavy path stores a pointer to its parent in the trie, or equivalently, the root of the tree of light children of that parent. We cannot afford to store this parent pointer, so instead we define the finger to ignore the intermediate nodes of the auxiliary tree, so that the parent of the current finger gives us the desired root. Nodes in an auxiliary tree also contain pointers to their maximum-weight leaf descendents; we replace these pointers with relative fingers. Sleator and Tarjan's link-cut trees order auxiliary trees by decreasing weight; this detail is unnecessary, so we instead order the children by key value to speed up navigation.

With these modifications, the link-cut trees of Sleator and Tarjan can be implemented functionally with path copying. To see that path copying induces no slowdown, note that although the finger makes jumps down, the finger is shortened only one node at a time. Each time we take the parent of a finger, the node at the end can be rebuilt from the old node and the nodes below it in constant time. Furthermore, because the maximum finger length is $O(\lg n)$, at the end of the operation, one can repeatedly take the parent of the finger to get the new root. Another way to see that path copying induces no slowdown is that the link-cut trees in the original paper also involved augmentation, so every ancestor of a modified node was already being rebuilt.

These functional link-cut trees let us support modification operations in $O(\lg n)$ time given fingers to the relevant nodes in our trie. It remains to see how to navigate a finger, the topic of Section 3.2, and how to maintain other fingers as the structure changes, the topic of Section 3.3.

3.2 Finger Movement. In this section, we describe three basic finger-movement operations: finding the root, navigating to the parent, and navigating to the child with a given label. In all cases, we aim simply to determine the operations required of our finger representation.

The root of the trie is simply the minimum element in the topmost heavy path, a finger for which is already stored at the root of the representation tree. Thus we can find the root of the trie in constant time.

The parent of a node in the trie is the parent of the node within its heavy path, unless it is the top of its heavy path, in which case it is the node for which this node is a light child. Equivalently, the parent of a node in the trie is the predecessor leaf in the globally biased search tree, if such a predecessor exists, or else it is the root of the auxiliary tree containing the node. We also defined the parent operation on fingers to solve the latter case. For the former case, we require a predecessor operation on a finger that finds the predecessor of the node among all ancestors of the node. If this operation takes constant time, then we can find the predecessor leaf within the globally biased search tree by taking the maximum descendant of the predecessor ancestor found by the finger.

A child of a node is either the node immediately below in the heavy path or one of its light children. The first case corresponds to finding the successor of the node within its globally biased search tree. Again we can support this operation in constant time by requiring a successor operation on a finger that finds the successor of the node among all ancestors of the node. Thus, if a child stores the label of the edge above it in the trie, we can test whether the desired child is the heavy child. Otherwise, we binary search in the auxiliary search tree of light children in $O(\lg \Delta)$ worst-case time. Because the total depth of a node among all trees of light children is $O(\lg n)$, the total time to walk to a node at depth d can be bounded by $O(d + \lg n)$ as well as $O(d \lg \Delta)$.

3.3 Multiple Fingers. This section describes how to migrate the constant number of fingers to the new version of the data structure created by an update, in $O(\lg n)$ time. We distinguish the finger at which the update takes place as *active*, and call the other fingers *passive*. Just before performing the update, we decompose each passive finger into two relative fingers: the longest common subpath with the active finger, and the remainder of the passive finger. This decomposition can be obtained in $O(\lg n)$ time given constant-time operation to remove the topmost or bottommost node from a relative finger. First, repeatedly remove the topmost nodes from both the active and passive fingers until the removed nodes differ, resulting in the remainder part of the passive finger. Next, repeatedly remove the bottommost node of (say) the active finger until reaching this branching point, resulting in the common part of the passive finger. Now perform the update, and modify the nodes along the active finger according to path copying. We can find the replacement top part of the passive finger again by repeatedly removing the bottommost node of the active finger; the remainder part does not need to change. Then we use the concatenate operation to join these two relative fingers to restore a valid passive finger. Of course, we perform this partition, replacement, and rejoin trick for each relative finger.

3.4 Finger Representation. Recall that each finger to a node must be a list of the ancestors in the representation tree of that node supporting: push and pop (to move up and down); concatenate (to follow relative fingers); inject (to augment relative fingers based on a node's children); eject (to allow for multiple fingers); parent outside path (for faster parent query); and predecessor and successor among the ancestors of the node (to support parent and child in the trie). Our finger must also be functional, because nodes store relative fingers.

As a building block, catenable deques support all the operations we want except for predecessor or successor. Moreover, functional catenable deques have been well researched, with Okasaki giving a fairly simple $O(1)$ method [Oka98] that is amortized, but in a way that permits confluent usage if one allows memoization. Furthermore, Kaplan and Tarjan have shown a complicated $O(1)$ worst case purely functional implementation of catenable deques [KT95].

In order to implement predecessor and successor queries, decompose the path in the representation tree into a sequence of right paths (sequence of nodes where the next element on the path is the right child) and left paths (sequence of nodes where the next element on the path is the left child). Then, the predecessor of a node among its ancestors is the last element of the last right path. The successor of a node among its ancestors is the last element of the last left path.

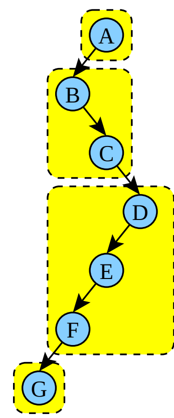

Fig. 3. A finger to G is represented by G and a deque of (outlined) deques of ancestors of G.

Instead of maintaining the finger as one catenable deque, we represent the finger as a deque of catenable deques, alternating right and left paths; see Figure 3. Then, because the last right path (or left path, respectively) is either the ultimate or penultimate deque in the sequence, its last element can be retrieved in $O(1)$ time. All other operations of the standard catenable deque can be simulated on the deque of deques with only $O(1)$ overhead. We thus obtain a structure that supports all of the operations of normal catenable deques of nodes, predecessor and successor in $O(1)$ time.

This suffices to describe the basic data structure for functional tries. Their query time is $O(\lg \Delta)$ for navigation downward, $O(1)$ for navigation up, and $O(\lg n)$ for updates.

4 Adding Hash Tables

Our data structures above take $O(\lg \Delta)$ to move a finger to a child node. We can improve this bound to $O(\lg \lg \Delta)$ by introducing fully persistent hash tables. The resulting data structures are confluently persistent, but no longer functional.

For our first structure (Section 2), we show in the full paper how to construct a fully persistent hash table on n elements that performs insertions, deletions, and searches in $O(\lg \lg n)$ expected amortized time. Using this structure instead of the balanced trees of neighboring vertices at every vertex, the time and space cost of updates and finger movements improves to $O(\lg \lg \Delta)$ expected amortized.

We can also use this method to improve our second structure (Section 3.1). Given a set of elements with weights $w_1, \ldots, w_n$, we develop in the full paper a weight-balanced fully persistent hash table with $O(\lg \frac{\sum_j w_j}{w_e})$ expected amortized time modification, $O(\lg \lg n)$ find, and $O(\lg n)$ insert and delete, where n is the number of elements in the version of the hash table being accessed or modified and w_e is the weight of the element being accessed.

To use a hash table to move a finger down the trie, we modify each node of our data structure to include a hash table of relative fingers to light children *in* *addition* to the binary tree of light children. The binary tree of light children is necessary to support quickly recomputing the heavy child on an update; this would be hard with just a hash table. Except for inserts and deletes, the hash table achieves at least as good time bounds as the weight-balance tree, and each trie operation involves at most $O(1)$ inserts or deletes, so we can maintain both the table and tree in parallel without overhead. As a result, updates still take $O(\lg n)$, moving the finger up still takes $O(1)$, and moving it down now takes $O(\lg \lg \Delta)$, where Δ is the degree of the node being moved from. The hash tables depend on fully persistent arrays, which are expected amortized, so updates and moving a finger down become expected amortized.

5 Open Problems

It would be interesting to combine our two functional data structures into one that achieves the minimum of both performances. In particular, it would be nice to be able to modify a node at depth d in $O(\min\{d \lg \Delta, \lg n\})$ time and space. One approach is to develop modified globally biased binary tree where the depth of the ith smallest node is $O(\min\{i, w_i\})$ and supporting fast splits and joins.

Acknowledgments. We thank Patrick Havelange for initial analysis of and experiments with version control systems such as Subversion.

References

[BST85] Samuel W. Bent, Daniel Dominic Sleator, and Robert Endre Tarjan. Biased search trees. *SIAM J. Comput.*, 14(3):545–568, 1985.

[DB99] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. *SIGMETRICS Perform. Eval. Rev.*, 27(1):59–70, 1999.

[Die89] Paul F. Dietz. Fully persistent arrays. In *WADS 1989*, LNCS 382, 67–74.

[DSST89] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[DST94] James R. Driscoll, Daniel D. K. Sleator, and Robert E. Tarjan. Fully persistent lists with catenation. *J. ACM*, 41(5):943–959, 1994.

[FK03] Amos Fiat and Haim Kaplan. Making data structures confluently persistent. *J. Algorithms*, 48(1):16–58, 2003.

[KT95] Haim Kaplan and Robert E. Tarjan. Persistent lists with catenation via recursive slow-down. In *STOC 1995*, 93–102.

[Mit03] Michael Mitzenmacher. Dynamic models for file sizes and double Pareto distributions. *Internet Math.*, 1(3):305–333, 2003.

[Oka98] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[PCD05] Dan R. K. Ports, Austin T. Clements, and Erik D. Demaine. PersiFS: A versioned file system with an efficient representation. In *SoSP 2005*.

[Pip97] Nicholas Pippenger. Pure versus impure lisp. *ACM Trans. Program. Lang. Syst.*, 19(2):223–238, 1997.

[PT07] Mihai Pătraşcu and Mikkel Thorup. Randomization does not help searching predecessors. In *SODA 2007*, 555–564.

[ST83] Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.