High-Performance Persistent Graphs

Storing graphs in wide-fanout key-mapped tries with lazy copying persistence

John Moody
Colorado College '16
john.moody@coloradocollege.edu

Benjamin Ylvisaker
Assistant Professor, Colorado College
ben.ylvisaker@coloradocollege.edu

Abstract

Persistent data structures allow large and complex data structures to be copied and manipulated inexpensively. The persistent way of representing data offers opportunities to more elegantly and more efficiently implement certain algorithms and programming patterns. Few persistent data structure libraries, however, are designed with an emphasis on speed and performance compared to their mutable cousins. We describe and present a C library for a persistent graph data structure, which uses array compression techniques and balanced wide-fanout tries to create a structure that enables persistence without sacrificing performance. Compared to a competitive C++ mutable graph library, we consistently achieve 30-40% slower random read performance using up to 30% fewer bytes in memory, with the benefit of highly space-efficient persistence.

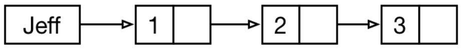
Keywords persistent data structures, graphs, hash array mapped trie

1. Introduction

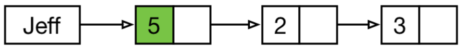
Graphs are one of the basic data structures that we use to represent a wide variety of data. A graph defines some number of nodes and edges, which connect nodes together. Graphs have wide-ranging applications, from computational models to databases, networking and pathfinding. To get information about nodes or edges in a graph within a program we typically use an array or some manner of key-value store. A graph node's value is usually a list of adjacent nodes, which are either predecessors to that node or successors. The value associated with an edge is typically just two identifiers for its predecessor and successor. This paper explores storing graphs as a persistent data structure. We will give a background on persistent data structures, tries, and array compression techniques from the Hash Array Mapped Trie, and then propose a structure for persistent graphs using those concepts that we will see has strong performance characteristics for various operations and highly efficient use of memory.

2. Persistence

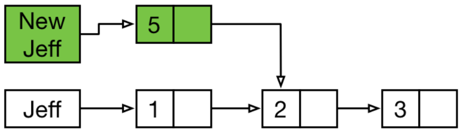
What does it mean for data to be stored in a persistent way? A piece of data is persistent if it does not change. Consider a linked list in memory, Jeff:



There are a number of ways to make an edit to this structure. If we wanted to change the frontal value of Jeff from 1 to 5, and we do not need the original any longer, we may simply change it:



If we want this notion of persistence to apply to Jeff, however, Jeff cannot change. Instead, we need to come up with a way to change the front value of Jeff to 5 while keeping the original version of Jeff with 1 at the front intact. Enter "New Jeff":



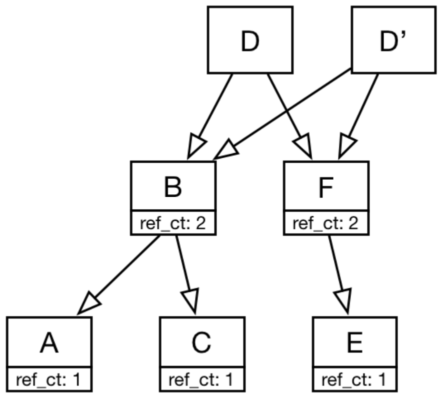
Jeff, we notice, has not changed. New Jeff preserves the parts of Jeff's structure that they have in common. Persistent data structures, then, refer to structures like Jeff and New Jeff, which, after being created, will always remain the same.

2.1 Why?

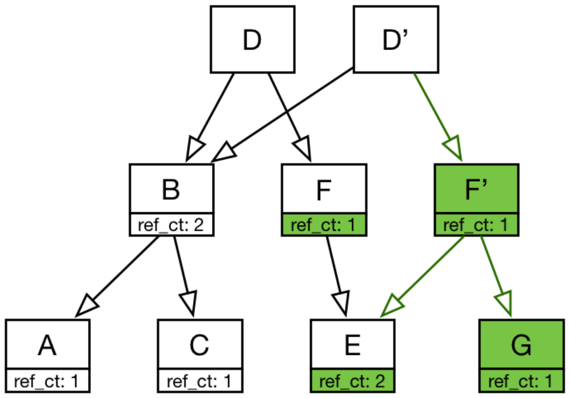
In a broad sense, persistent data structures offer a way to do quick, cheap analysis on multiple versions of a large data structure. In a mutable system, analyzing multiple versions of a data structure typically involves expensive wholesale copying of the structure, with no easy way to reverse changes that have been performed. With persistent data structures, copies of a structure can be very small in size relative to the entire structure. Reverting changes to a persistent structure is simple, since the earlier version can still be accessed and the new version easily deleted, as in the above example. In some kind of GUI application such as Photoshop, or a 3D modeling program, this persistent model can reduce the complexity associated with implementing an "undo" functionality, since such changes can be made inexpensively and simply to even a complicated structure that might define a 3D model or other construction. Further, if systems that perform analysis on large data sets are concerned with change to that data set over time, persistent data structures can be a powerful tool both in terms of efficiency, both in terms of memory and write performance. Persistent data structures are also guaranteed to be thread-safe, since operations on persistent structures will never write to the parts of their structure that they share with other versions. This characteristic makes persistent data structures more easily and safely manipulated by multiple processes and threads simultaneously. The preponderance of consumer-level multi-core processors makes these performance gains more generally available.

2.2 Trees and Reference Counting

Let us now consider an example using a simple binary search tree, where we have D, and a separate copy D':



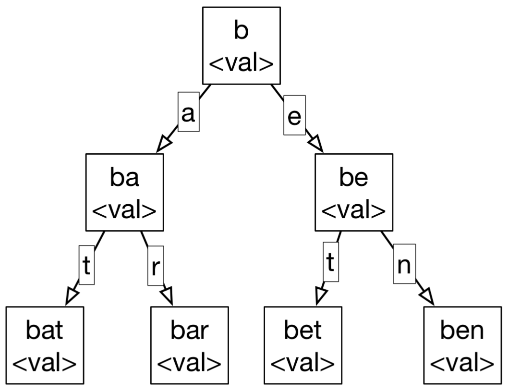
For us to be able to make edits to D' without changing D, we must introduce the concept of reference counting. A reference count keeps track of how many objects point to a given node. Here, since B and F have reference counts greater than one, we know that we can't modify those nodes without changing another version of the data structure. Therefore, when we insert G into D', we will copy any nodes that have reference counts greater than one and adjust the tree as necessary:



The same concept applies in deleting or modifying nodes.

2.3 Tries

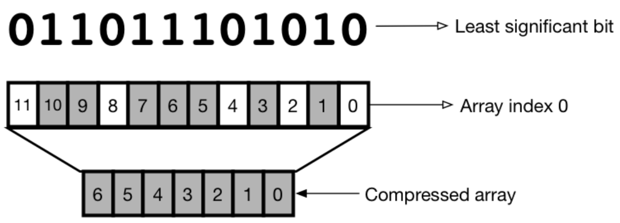
The trie [tɹaɪ] is a manner of key-value store which takes the form of a tree:



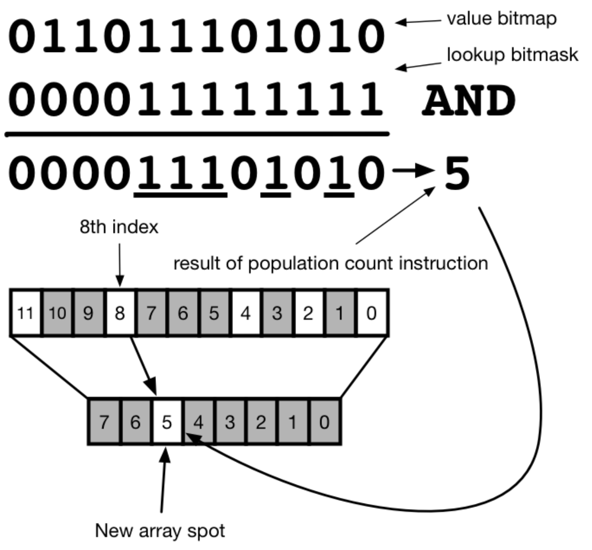
In a tree data structure, we find values by comparing them to values stored at higher levels and deciding what branch to descend into. In a trie, values are looked up by keys, which define their unique location in the trie. In this figure, the value associated with "bet" is defined by its unique key: "`b-e-t`". This system enables fast lookup proportional to the length of the key. Our structure will bear significant resemblance to this trie, but with some modifications.

2.4 Array Compression

Later, when we introduce our data structure, we will need to represent a sparse array using the least amount of memory possible. We will discuss here the array compression scheme that we use, which is borrowed from Phil Bagwell's hash array mapped trie:



We here use a bitmap to represent which spots in the sparse array (indices 2, 4, 6, 7, 8, 10, and 11) are currently occupied by values. Let's imagine we want to check what's at the eighth index of the hypothetical array. To verify whether this spot is empty, we will first bitwise `AND` together the bitmap and a bitmask solely composed of zeroes, with a one occupying the eighth least significant digit. Since the bitmap contains a zero at that digit, the result of that arithmetic will be zero. That result confirms that there is a free spot in the array at that index. Now, if we want to insert a new value into our compressed array, we need to find out what spot in the compressed array we must insert into. To do so, we will bitwise `AND` together the bitmap and a number whose eight least significant bits are ones.



The number of ones in the result represents how many spots are occupied in the compressed array prior to the one we want to insert into (or, in other words, the array index of our desired spot). We will now use the population count instruction, which is built in to most modern processor architectures and returns the number of ones in a binary number, to derive the index we need in our dense array, five.

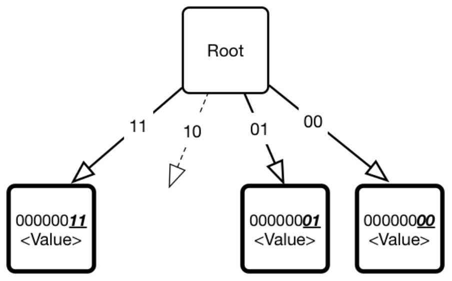
Using this scheme we conserve the memory that would normally be occupied by empty array slots. This conservation is particularly important if we are updating this array persistently, since creating copies of empty array slots would introduce large amounts of waste.

3. Wide-fanout key-mapped trie

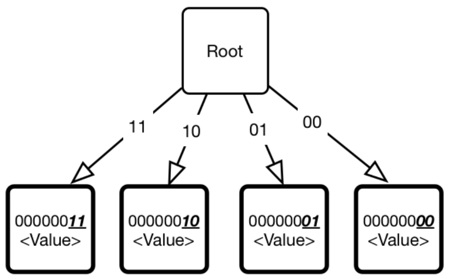
Armed with a knowledge of persistence, tries, array compression, we may now introduce the structure that our library employs for persistent graph storage. Our structure has the following characteristics:

- Each value is given a unique numeric key either during or prior to insertion according to the current balance of the trie, which defines its position in the trie.
- Valus stored only in leaves.
- Wide fanout, to minimize trie depth and key length.
- Array compression, with bitmaps to indicate non-null positions.
- Values chunked together with their parent nodes.
- Nodes without children combined with their parents, to reduce the number of pointers.

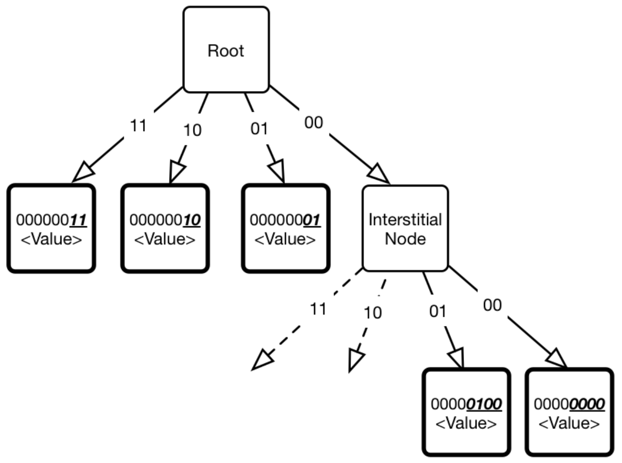
In our trie example from earlier, the branching factor was defined alphabetically; at each node, the trie had 26 options on branches to pursue. Here, we can define the branching factor of our trie by defining how many digits of the numeric key we are assigning at each level. To understand how this works, let us consider a hypothetical insert with keys of length 8, considering two bits at a time. Since we are considering two bits of the key at a time in determining which branch of the trie to pursue, our fanout is four:



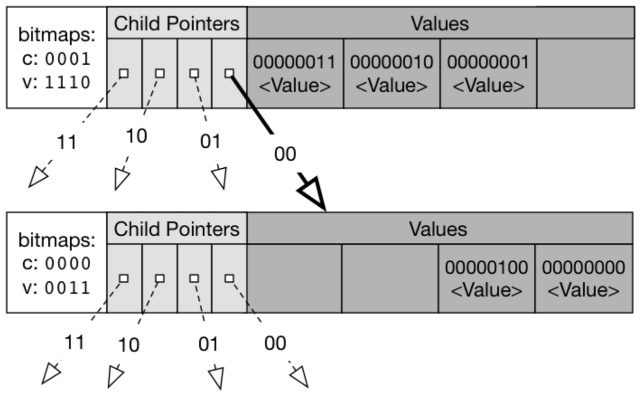
Here, we see that there is a free spot in the root at `10`. We will insert the value here.



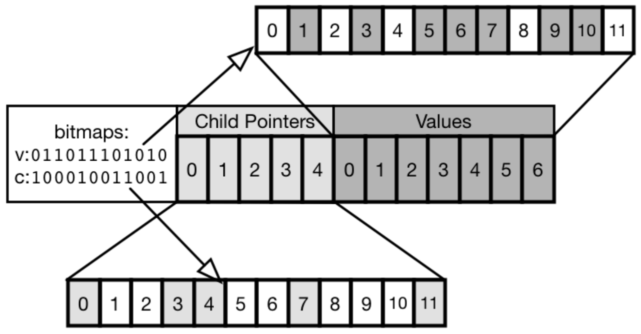
For our next insert, there are no free spots for values, so we will create a new child:



When we look up one of our nodes by the key, the library will examine the first two bits of the key, `10`. Further, since all values are stored in the leaves of the trie, we will actually store the bolded values in arrays at the tail end of each parent node. Hence, our current trie will actually appear like this in memory:

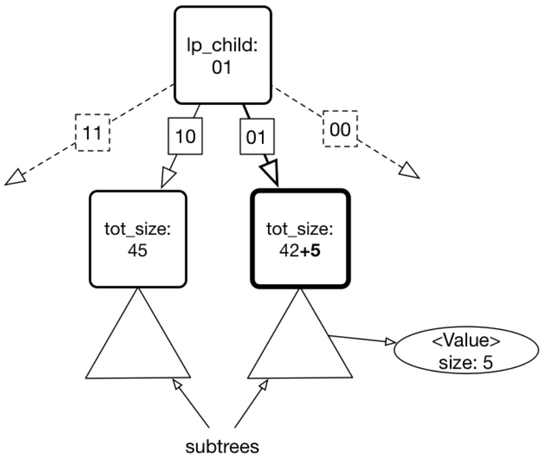


Our pointers to children are stored in arrays, as well as values that occupy spots directly beneath nodes. For a fan-out of four, the empty spots in these arrays will not waste very much memory. If we are examining five bits of the key at a time, however, the branching factor of our trie is 32. Representing sparse arrays of length 32 in memory and copying them persistently will waste massive amounts of memory if the arrays are mostly empty. Hence, we will use our array compression scheme from earlier to represent our arrays. Here is a complete picture of a hypothetical node in our structure, with sparse arrays and bitmaps of size 12.

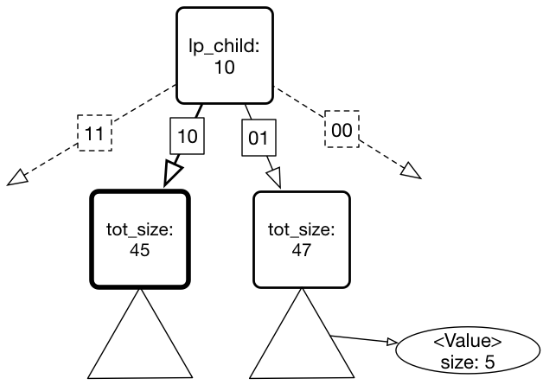


3.1 Trie Balancing

To ensure good asymptotics when looking up values at random in our graph, as might happen during a traversal of the graph, we want to keep our trie as balanced as possible. 'Balance,' in the context of tree-like structures, means that values are equally distributed in branches. If the trie is uniformly deep, we have a uniform lookup speed for all values. For our trie to retain balance, we have to create keys for new values that place them in the appropriately least-populated section of the trie. We employ a simple scheme which could be optimized for greater performance. In this scheme, a node stores the total size of everything beneath it in the trie structure. Until a node is found with a free spot for a value, the library descends into the least populated branch of the trie by checking the total sizes of all available children. After each discrete insertion or deletion from our trie, the total size value of the parent is adjusted in accordance with the difference between the child's old size and the size after insertion or deletion. In this figure, a value of size 5 is inserted into the right-most branch of the trie, which has been found to be least populated.



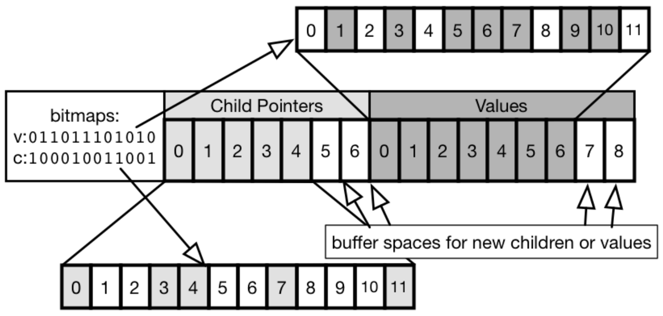
After the insert is performed, the total size value for the right child is adjusted.



At the next insertion, the library will instead find the left child to be least populated, and will descend into that branch for insertion.

3.2 Buffers

With our nicely packed arrays, every time we wish to insert into our trie structure, we need to resize the array of the node we insert into. Since structs in C are not dynamically sized, the addition of a new value necessitates re-allocating the entire struct instance to compensate for the newly resized array. This results in huge amounts of churning memory, which means very expensive writes. To reduce the complexity and memory churn of the average write, we introduce `n`-sized buffers at the end of our dense arrays. These buffers, which may be resized by the application, represent free space in which a node may store `n` values or children before the struct instance will be full and in need of reallocation. This significantly decreases the amount of memory being allocated and freed on each insert on average. The exact effects of the various sizes of these buffers are detailed in our Results section. Revisiting our diagram from the previous section, we can see how the buffers behave in our implementation:



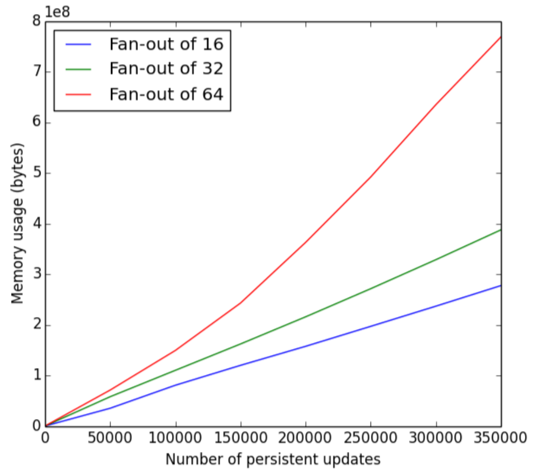
3.3 Lazy Copying

Next we will discuss the exact flavor of persistence that our library implements, "lazy copying." Using lazy copying, an update to a trie structure `n` will, by default, be performed in place, without preserving the previous version. Creating a copy of `n` causes a copy of `n`, `n'`, to be created, and increments the reference counts of its children. Then, if an update to to either `n` or `n'` changes a node with a reference count greater than one, that node is first copied to prevent other versions of the structure from being modified. A high-level example of how this works is seen in the earlier Trees and Reference Counting section.

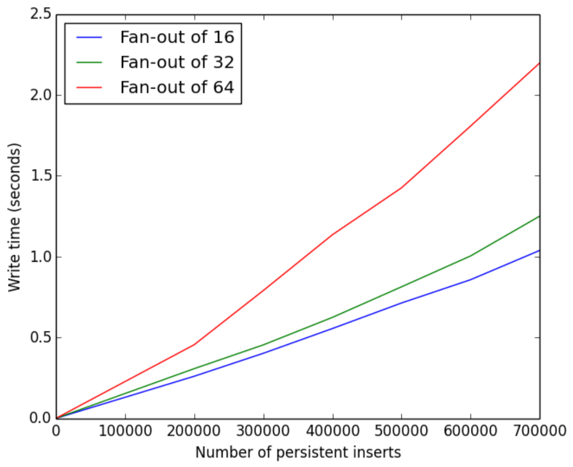
Chunking together values with their parents will lead to redundancy under lazy copying, we notice. If an update to a branch of the trie requires that branch to be copied, and the nodes in that branch are full of values, we will make many redundant copies of the values in that branch. We consider this an acceptable tradeoff considering the advantages in read-performance achieved through chunking. Chunking reduces the number of pointers pursued for each lookup by one. Since our trie has max depth of five under a 32-fanout configuration, this difference is significant, though we do not provide results here that compare chunk and no-chunk configurations.

4. Results

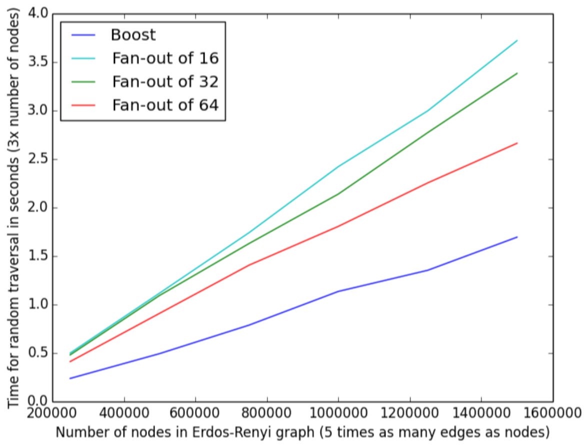
We include tests that show our memory usage and write performance for persistent updates under different configurations of the library. Next, we include a comparison of write performance and memory footprint for different values of `VALUE_BUFFER_SIZE` and `CHILD_BUFFER_SIZE`. Lastly, for the time complexities of a random traversal of a randomly generated Erdos-Renyi graph, we compare to the Boost Graph Library included in C++.



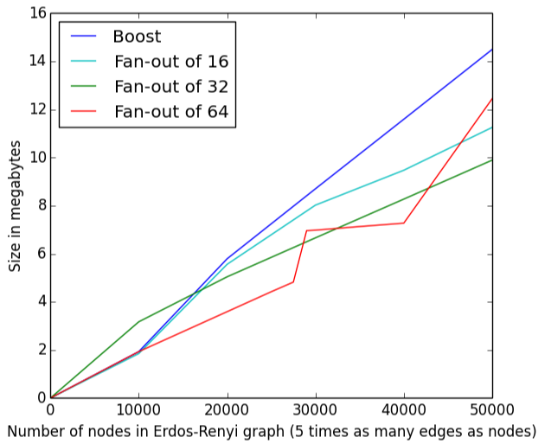
Our results here may seem unintuitive; for lower fanouts, our tries are deeper on average. Hence, a persistent insert would require more copies to insert at the bottom of the trie. However, since our values are chunked together with parent nodes, every copy of a node in the 64-fanout trie copies more values. Thus, our 64-fanout configuration uses more memory for persistent updates.



Write time increases in proportion to the amount of memory copied.

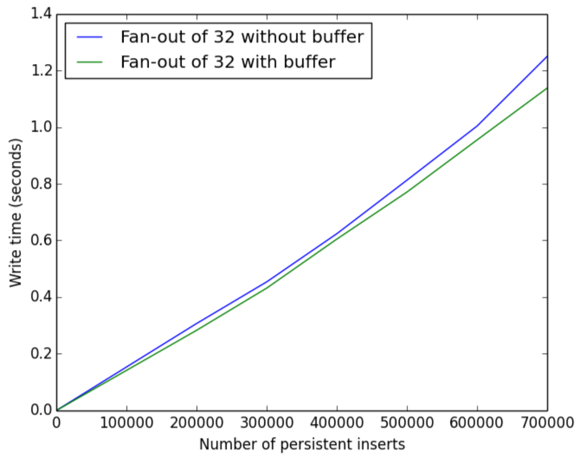


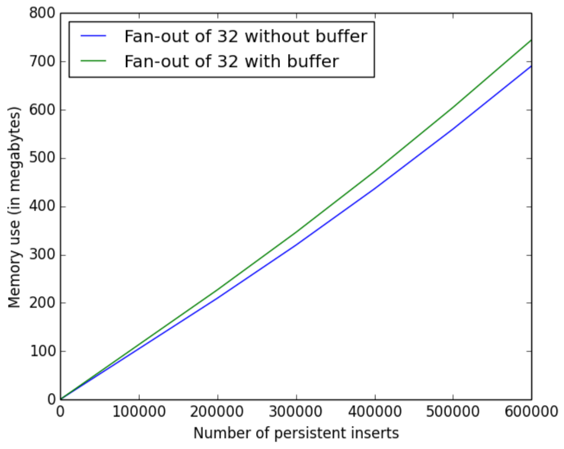
As predicted, with shallower trees, our reads are faster. Compared to the Boost graph library, we are between 30 and 50% slower.



For a single graph, our implementation is usually more memory-efficient.

The following two graphs compare write performance and memory usage for a branching factor of 32, with and without a single value buffer spot:





We anticipated that our implementation of the value buffers at the end of arrays would substantially increase write performance while having a minimal impact on memory footprint. We suspect that with a better implementation, we can achieve better performance. We saw a write performance increase only with a value-buffer of size one, in the 32-fanout configuration. All other buffer configurations saw a negative impact on write performance as well as memory footprint. Further research can shed more light on how a more solid implementation of these buffers can improve write performance while having a minimal impact on memory footprint.

References

Bagwell, Phil (2000). Ideal Hash Trees.