Fast Type-based Indexing and Querying of Dynamic Hierarchical Data

Master's Project Report

Sam Johnson <samuel_kelly@brown.edu>, Sc.M Brown University '17

Introduction

Over the course of this project I describe and implement (in a library called Hierarch) a novel indexing scheme and search algorithm for typed, in-memory trees that is robust to frequent modifications and is capable of answering type-based containment queries (e.g. "get descendants by type" queries) in near-constant time depending on the distribution of node types and verticality of the tree. I then benchmark variations of my algorithm against the naïve tree traversal approach commonly used to answer these queries in static analysis and similar domains, and I make some educated recommendations for which variation of my algorithm to use in general. A major contribution of this work is the introduction of a novel data structure, the DFI Filter (Depth First Index Filter), which allows for efficient tracking and updating of the pre-order indices of nodes in a tree that is subject to deep structural changes, regardless of tree sparsity or branching factor. In order to deal with types, I extend this data structure so that it can provide an effective mapping between the pre-order index space of a large, typed tree, and the pre-order index spaces of each of its component types, giving my algorithm the ability to rapidly find the number of nodes (or return an iterator over these nodes) that are descendants of the specified tree node. This work is of particular importance in light of a recent empirical study I conducted, which found that over 58% of the selector queries issued in the JavaScript language could be sped up by at least one order of magnitude using the Hierarch approach of

constructing a dynamic index capable of answering get-descendents-by-type queries. I conclude that the embedded-passive approach results in the best overall return in terms of general purpose read and write performance.

Overview

A get-descendants-by-type (GDBT) query is a typed-tree query of the form “given an arbitrary tree node A and an arbitrary node type T, find all descendants of A that are of type T. While often overlooked in the existing literature, GDBT queries nonetheless appear in a wide variety of domains and sub-fields of computer science [1], including but not limited to compilers, static analysis, databases, file systems, machine learning, search engines, operating systems—essentially anywhere that typed hierarchies or trees are regularly constructed, traversed, queried, or manipulated in some way. In fact, I have come to hold the high-level belief that the vast majority of non-trivial exploration and analysis that occurs in practice over typed hierarchical data can be expressed strictly as a composition of nested GDBT queries. This belief was recently affirmed by the results of my empirical study of the JavaScript language [2] which found that over 58% of DOM-based selector queries are pure compositions of GDBT queries, and that 100% were compositions of GDBT queries and a small set of trivial operators[1]. So far I have seen little reason to doubt that these results generalize to other areas where typed tree querying is important, as I have also seen tremendous performance boosts in compilers/static analysis, and file system indexing.

Over the course of my Master’s project, I developed a high-speed C++ implementation and benchmarks that cover all four variations of my indexing algorithm, including active and

[1] preliminary results can be found at https://github.com/samkelly/hierarch/tree/master/empirical_study

passive successor tracking as well as lazy and embedded index management. My ultimate finding was that the embedded index management scheme combined with passive successor tracking is best for most workloads, whereas the embedded index management scheme combined with active successor tracking is best for very read-heavy workloads. I also performed new benchmarks that re-establish the general get-descendants-by-type technique as an order of magnitude faster than the naive approach of manually traversing each subtree.

Since early 2013, I have been developing various indexing algorithms that specifically seek to optimize GDBT queries, with the ultimate goal of creating an algorithm that can fully index a typed, dynamic[2], roughly balanced[3] tree in linear time and linear memory, but can also answer GDBT queries and react to structural changes (leaf node additions or deletions) in constant time. In perhaps the only published work on the topic, [1], I present variations of an algorithm capable of solving the GDBT indexing problem in linear time and linear memory, and the GDBT querying problem in amortized constant time by exploiting per-type pre-order depth-first index properties. However, this approach is not robust to structural changes and requires a full $\Theta(n)$ re-indexing operation upon even the simplest of tree modifications, making it inviable for use with *dynamic* trees and domains where tree changes are at all frequent or expected. Indeed, much more work is required to be able to achieve similar performance with dynamic trees.

Since the 2014 publication [1] I have devised a much more complex indexing algorithm using a novel data structure (the DFilter)[4] that both answers GDBT queries and responds to

---

[2] *dynamic* in the sense that the tree can be modified such that nodes are added or removed. Strictly speaking I define this as the addition or deletion of a single leaf node since one can represent any tree change as a sequence of leaf node additions and deletions.

[3] a linked-list is technically speaking a tree, but is not interesting for GDBT queries as there is no branching. Similarly, a single root node with many children but no further descendants is also not interesting for GDBT queries. Typed trees that are roughly balanced (with a depth roughly close to log(n)) *are* interesting, and are the focus of this paper.

[4] source code, results, and discussion can be found at https://github.com/sam0x17/hierarch

structural tree changes in near-constant time (in practice, effectively constant time). Unlike its older cousin, this much more powerful algorithm is suitable for use in areas and domains where typed tree modifications are frequently interleaved at runtime with type-based queries and searches, including but not limited to file systems, databases, graph search, static analysis, and the DOM in modern web browsers. This approach involves connecting the nodes of a global AVL tree whose keys roughly correspond with the depth-first indices of nodes in the main tree, with a conglomerate of interconnected per-type AVL trees whose keys roughly correspond with the type-wise[5] depth-first indices of nodes in the main tree, and efficiently updating and maintaining these connections when tree nodes are added or deleted.

Previously, I completed the aforementioned empirical study on the JavaScript language, and completed a partial implementation of the lazy-passive version of the DFilter algorithm, which is the simplest to implement. This semester I was able to implement and benchmark all four variations of the algorithm (lazy-passive, lazy-active, embedded-passive, and embedded-active), and build a C-API that could easily be connected via native extensions to higher level languages and environments, such as Node.js using node-gyp and Ruby using Ruby FFI. I painstakingly designed the API such that native extension implementers can completely avoid dealing directly with dynamic memory allocation, instead allowing them to call simple functions to create, destroy, and switch contexts (a context is the active tree being operated on by the API) and identify nodes based on unique IDs that are context-specific. The most (general-purpose) performant variant of my algorithm, embedded-passive, has been left available in the default master branch of the project, whereas the other variants and the rest of the benchmarking code can be found spread out across the other branches.

[5] i.e. the depth-first index if we consider only nodes of type T during our traversal

Background

The main insight behind my algorithm comes from the inherent properties of a tree that has been labeled with the indices its nodes would have during a pre-order depth-first traversal. In [1], I derive the **depth-first indexing theorem**, which states that, given a node A, and the node's *successor*, B (the next node that will be traversed once all of the children of that node have been traversed), the pre-order depth-first indices of all descendents of A lay on a critical strip between A's pre-order depth first index, and B's pre-order depth-first index. Thus if the only thing we know about a node is its pre-order depth-first index, we can determine that it is a descendent of A merely by performing an $O(1)$ range check to see if it lays within A's critical strip. Likewise, if you have a list of nodes sorted by their pre-order depth-first indices, you can perform two $O(log(n))$ binary search operations on the list for A's and B's indices as a quick way of finding the sub-list containing only descendents of A. If we replace this sorted list with a more resizable data structure, such as a binary search tree, or in our case, an AVL tree, then we can grow and shrink the number of nodes in our "list" inexpensively, all the while with get-descendents-by-type at our fingertips thanks to the depth-first indexing theorem.

A foundational paper in the field of XML indexing [3] anticipates the importance of pre-order depth-first indices, but laments their strictly sequential, and thus theoretically intractable nature. The problem is if you have one hundred nodes, with depth-first indices one through one hundred, and you want to delete node sixty-six, there will now be a gap in the depth-first indices unless we bump down all thirty-four nodes that come after node sixty-six by one. Likewise, if we want to add a node to be a child of node sixty-six, we must bump all subsequent nodes up by one. Because of this annoying property, it has been thought for years

[2][1] that it is virtually impossible to do anything useful with pre-order depth-first indices without incurring an expensive $O(n)$ re-indexing operation.

Hierarch overcomes this limitation in two ways. In the “lazy” case, Hierarch uses a caching system of offsets, where every node has a “mod” (modification ID) and a copy of its parent’s index. Any time a change is made to the structure of the tree resulting in an index being deleted or displaced, this information is bubbled up from parent node to parent node until it hits the root. Differences are calculated (and thus updates are applied) based on the discrepancy between the node’s cached version of the parent index, and the actual parent index. This scheme is simple to implement, but suffers from lower performance on highly vertical trees, since in the worst case, a completely straight-down linear chain of nodes, any change near the bottom must be propagated all the way to the root, every time.

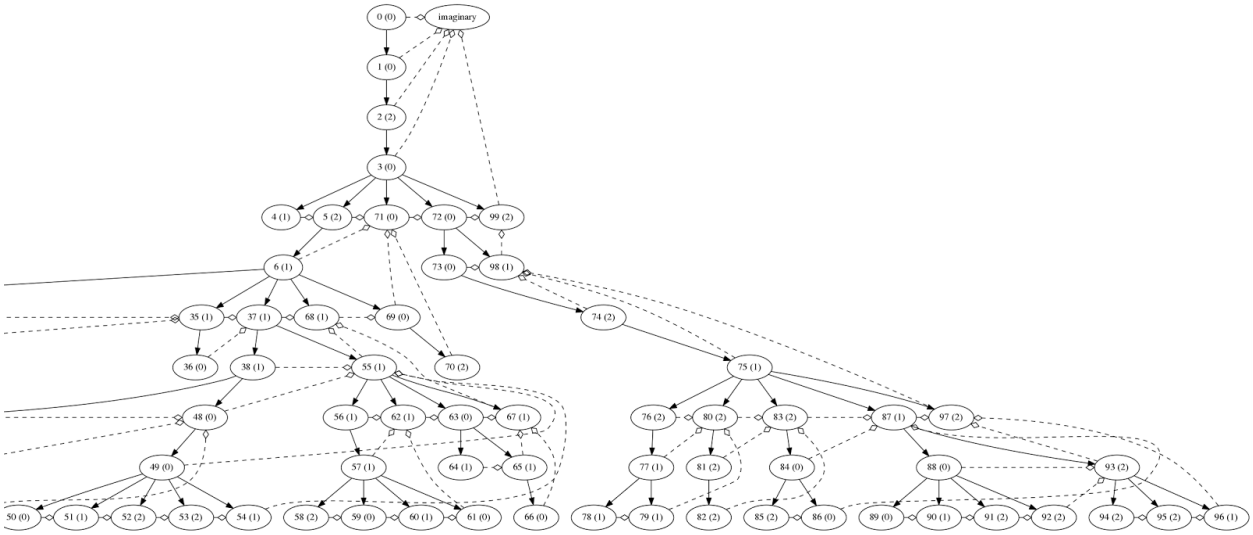
In the “embedded” case, Hierarch leverages the dual nature of every Hierarch node in the tree, namely that every regular tree node in Hierarch is also an AVL tree node in the global AVL tree that is sorted by depth-first index. Thus in this approach, we store an “offset” attribute and a “base_index” attribute on each node, as well as a “mod” (representing the modification ID once again). If we want to know the true index of a node that is out of date, we bubble up following AVL parents until we hit an up-to-date node, then we take the same path back down, adding up the cumulative offset as we go, and propagating this offset down one level on any nodes that have an index affected by the offset. This process is significantly more difficult to implement correctly (it took many more months than the other approach) but has the theoretical benefit of removing the linear worst case that occurs when we use the real-tree parent nodes instead of the AVL parent nodes for propagation — the farthest we will ever travel to update a pre-order depth-first index is the current height of the AVL tree, which will never exceed $O(log(n))$. Thus even in the case of a pathological “straight down linear line” tree, it only takes

$O(log(n))$ steps to verify and/or update the depth-first index of a particular node, which is rather remarkable considering how pathological a straight-down tree really is.

Successor Tracking

Another equally important task that must be carried out in a system that uses the get-descendents-by-type trick like Hierarch does is the tracking of these successor nodes, or rather, for each node, we need to at any given point be able to easily determine the node that would be visited in a pre-order depth-first traversal after *all descendents of our node have been visited*. This requirement re-introduces an unfortunate trade-off, in that to be able to track successors, there must exist some sort of worst case where we perform a potentially linear operation to update the successor of an extremely long vertical linear chain.

The diagram below shows a portion of a Hierarch-labeled tree with successor links shown in dotted lines, types shown in parenthesis, and depth-first indices shown as the primary number on each node. The “imaginary” node is the default successor for any nodes that do not have a successor. If we look at the depth-first index of a node, and the depth-first index of its successor, these two indices should fully encapsulate the indices of all descendants.



The first approach to successor tracking, active successor tracking, is the most obvious but is more difficult to implement. Every node stores a link to its successor, and an array of links to its current predecessors (nodes that list this node as a successor). At write time (when we are adding or removing a leaf), we use the new leaf's parent's successor as the initial successor for the leaf. We then loop over all predecessors of this successor. If the predecessor's index is greater than the new leaf's parent's index, and is less than the leaf's index, then this predecessor's successor link is remapped to point to the new leaf node, and the predecessor is added to the new leaf node's set of predecessors and removed from the parent's set of incoming predecessors. This set of predecessor nodes re-introduces our age-old problem of highly vertical trees causing expensive operations — the worst case is a situation where we have a root node with a left child that has a long vertical chain, and a right child that is a leaf. If any changes are made to the long chain, the right child must update any predecessor links accordingly, which is $O(n)$ in the size of the vertical chain. The trade-off is that at read time, every node always has an up-to-date successor link, so in some cases it might be worth it to use this approach if reads far outnumber writes.

The other approach is passive successor tracking. This approach, which is analogous to passive index management, leans on our existing caching system and the simple algorithm for finding a successor with zero information to be able to track successors with much less bookkeeping, consistent overhead during write time, and amortized very minimal to no overhead during read time. The idea is to at each node store a link to the successor, and a cached copy of the successor's index. We initialize the successor of a newly-inserted node by applying a simple traversal algorithm (go up until you can go right) to the non-AVL portion of the tree. This process will once again have the same bad worst case on a completely vertical chain, since we have to follow the chain all the way up until we reach the root or find a node where we can go

right. Successors can also change if a new node is added between a node and its successor. If we look at a node's successor and find that the successor's index is out of sync with our cached copy of this index, then we know such a shift has occurred and that we must run the simple algorithm again to calculate the (potentially) new successor. This is a rare occurrence, and will only happen when we insert a node between a node and its successor, and then later go to ask the node for its index.

It is also worth noting that while the "simple algorithm" is expensive in the case of highly vertical trees, real-world trees tend to be much wider than they are tall [1][4], and in practice, a node's successor is almost always within three to four hops from the original node [4] via the simple algorithm. This is why despite its seemingly unattractive theoretical features, the passive successor tracking approach greatly outperforms the active approach in the majority of general purpose workflows.

Type Handling

Up until this point we have only alluded to how one could extend the essentially one-dimensional model described above to handle trees (that we actually care about) that have a notion of node "type", for example abstract syntax trees, HTML DOM, file systems, XML, search engines, databases, etc. This capability is essential if we want to be able to answer get-descendents-by-type queries, that is to find, given a node A, all nodes that are descendants of A and are of a specified type T. The extension is fairly simple and works the same way regardless of which variant of the indexing algorithm (or which successor tracking method) we use. Since a node can have multiple types, for each type for that node, we maintain an AVL node that exists in a type-specific AVL tree, such that there is exactly one type AVL tree for each unique type represented in the master tree. The type-specific AVL trees, like the global

AVL tree used by the master tree, are sorted by the pre-order depth-first indices of the individual nodes in the tree, however these depth-first indices correspond not with the tree that would be formed if we pruned all nodes not of that type from the master tree, and filled in the broken links. In theory, we could use the master tree indices exclusively here, however then we would lose the ability to know the exact count of the elements in our returned get-descendents-by-type queries without having to iterate over the result. Thus we must maintain type-specific indices as well as the master tree index and successor link.

Once this machinery is in place, we can answer a get-descendants-by-type query on a particular node by calculating the the node's master index, finding its successor, calculating the up-to-date index for that successor, and then performing two binary searches on the type specific AVL trees that apply to this query — one search that finds the first node greater than or equal to the node's master tree index, to get a lower bound, and one search that finds the last node less than the successor's master tree index, to find an upper bound. If the current node and its successor happen to be of the search type, then one or both of these binary searches can be omitted. Then a pre-order traversal starting at the lower bound and ending on the upper bound will exactly return the nodes of the desired type that are descendents of our starting node. In this way, type-specific AVL trees provide us with a filtering capability that serves as a uni-type window into a many-type tree. Links back to and from the corresponding master tree nodes allow us to do additional powerful things, for example given two up-to-date nodes, we can in $O(1)$ determine if either node is a descendent of the other using some simple range checks.

AVL Rebalancing

While the lazy index tracking scheme is unaffected by this, the embedded-style algorithms, since they rely on offsets that correspond with the exact physical structure of the

underlying master and type-specific AVL trees at any given point in time, are highly susceptible to corruption from automatic AVL rebalancing operations. In fact, this problem was the original reason it was so much harder to implement the embedded scheme, and is the main reason why the embedded style was not available in the original implementation of Hierarch.

To deal with automatic AVL rebalance operations, we inject code into the C library(s) used by Hierarch (both GNU AVL, and AvlTree are used depending on the algorithm variation). Whenever a portion of the tree is about to be rotated, we call $avl\_touch(node)$ on the affected nodes to ensure that no offsets are present within that region of the tree once the rotation proceeds. As long as there are no offsets immediately in the area being rotated, no corruption will occur.

Benchmarking

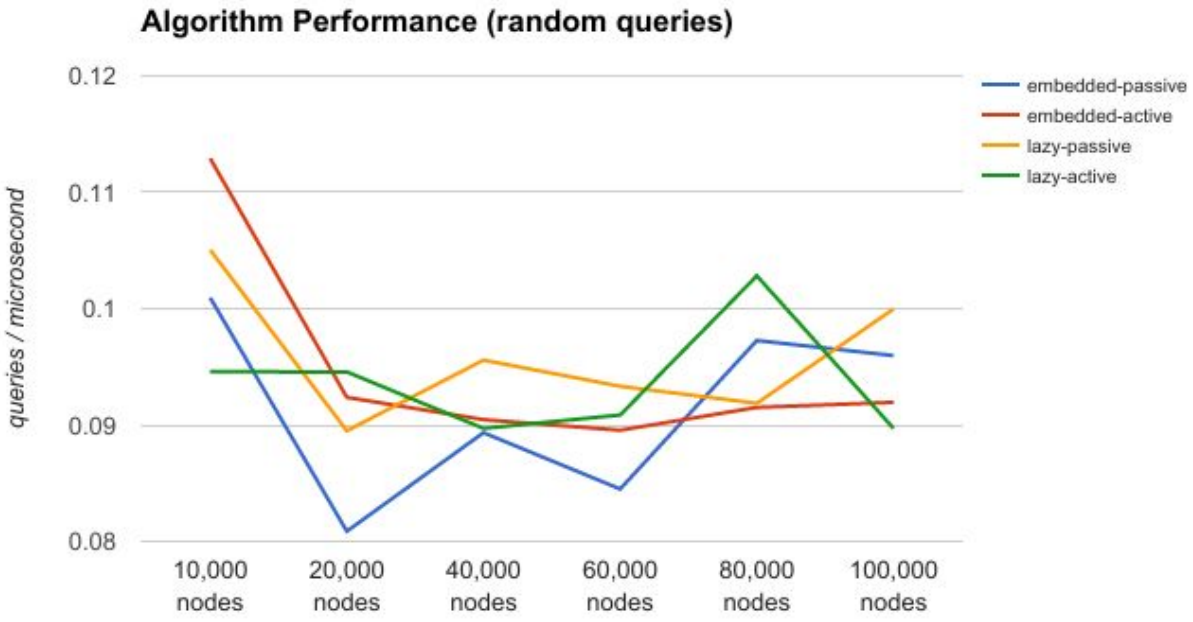
With a few tweaks to the code, the full benchmarking suite can be run, which compares the four algorithm variations against each other, and pits them against the naive approach. Here, it is useful to have an x-axis that corresponds with the number of nodes in a tree, however finding a sequence of successively larger trees is quite difficult, so synthetic data has to be used. Using statistics provided by the ROSE compiler team at Lawrence Livermore National Laboratory [4], I was able to produce as close to real-world as possible synthetic trees (matching typical abstract syntax trees for Java, C, and C++ respectively), and combining these with statistics on the average branching factor, and type distribution of nodes in the HTML DOM from the top 500 websites in Alexa. The tunable synthetic tree generation code can be found in v1/test.cpp and was used to produce most of the benchmarks.

Notably, I also was in talks with a local prominent SaaS company to use an anonymized copy of their database as “CRM” benchmarking data for Hierarch, however this deal and the

code associated with it fell through when the company realized that even when fully anonymized and viewed in aggregate, statistics on the hierarchical distribution of data in a database can reveal a lot about an organization, and can be used to estimate things like total number of active users, etc (even if we don't know which column corresponds with users, this is easy to figure out based on the relationships and numbers of records of different types). Another issue that plagued this sub-project was the difficulty of mapping a CRM database schema into a consistent tree-based view. Such a database is more readily represented as a graph, as there is no obvious node we could call the root, and there are cycles and DAGs embedded in the relationship between nodes. Still, with DAG resolution and a bit of creative pruning, this could have been a fascinating dataset to benchmark against, so it is really a shame the company in question decided to withdraw from the deal.

Benchmark Results

I evaluated each of the four Hierarch algorithms, lazy-passive, lazy-active, embedded-passive, and embedded-active against each other in terms of random node insertion and random node querying speed, using our synthetic tree model (which is an average of the statistics from Livermore and the Alexa data). Below, the 4-way comparison between the algorithms is shown for performing random get-descendents-by-type queries on synthetic trees of increasing sizes. For each data point, 100,000 queries were performed, and the average queries performed per microsecond is displayed on the y-axis.



As you we can clearly see from the graph, none of the algorithms seems to be bogged down by the fact that the number of nodes is increasing (this is good, as it supports our argument that these algorithms offer an in-practice constant-time solution), however since each algorithm was run on the exact same random trees as the other algorithms, we can also compare each algorithm's relative performance. From the graph, it can clearly be seen that the embedded-passive scheme has the best overall performance when it comes to random reads (this test ignores insertion/deletion performance). This is likely because the passive successor tracking scheme has very little bookkeeping and only hits its worst case behavior in the first hundred queries or so (after which point, all nodes are typically up-to-date). Likewise, the embedded index tracking scheme is complex at insertion/deletion time but very simple at read/query time, so its performance here really shines.

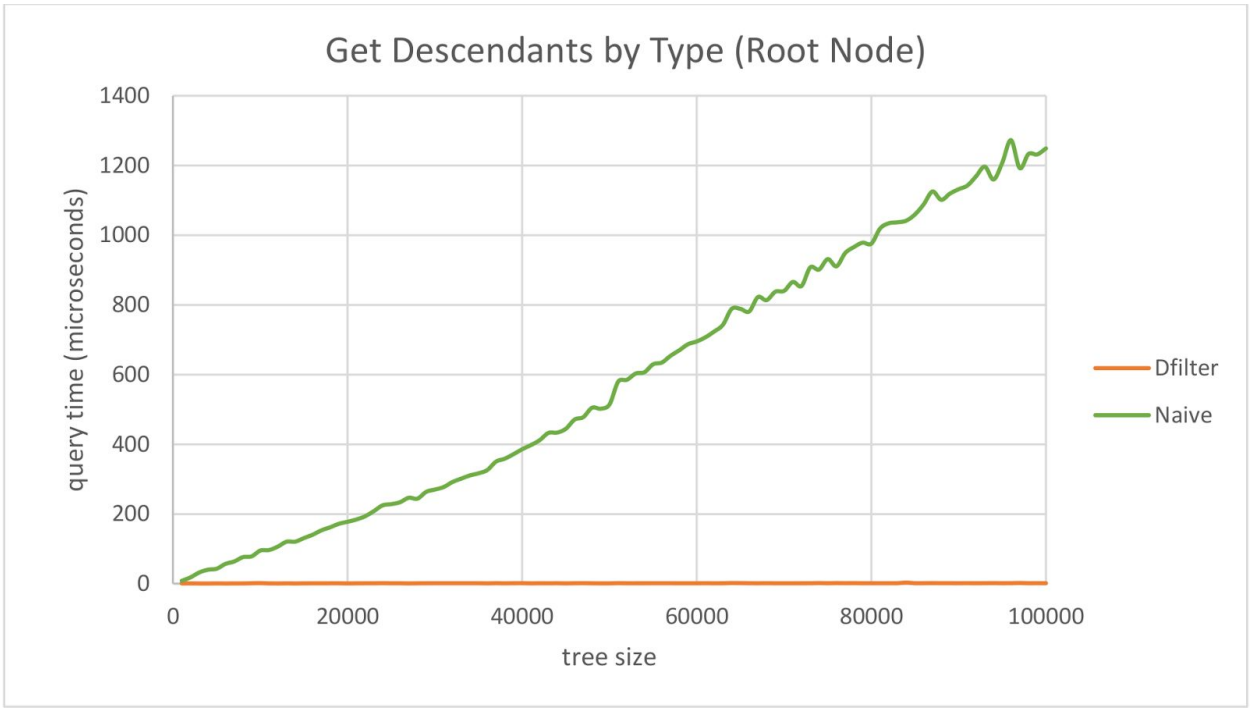
For insertion/deletion performance, all four variants performed very similarly, so we do not show a graph here, however it is worth noting that embedded-passive was once again the fastest despite the tradeoffs mentioned above. It is clear that the theoretical advantage of

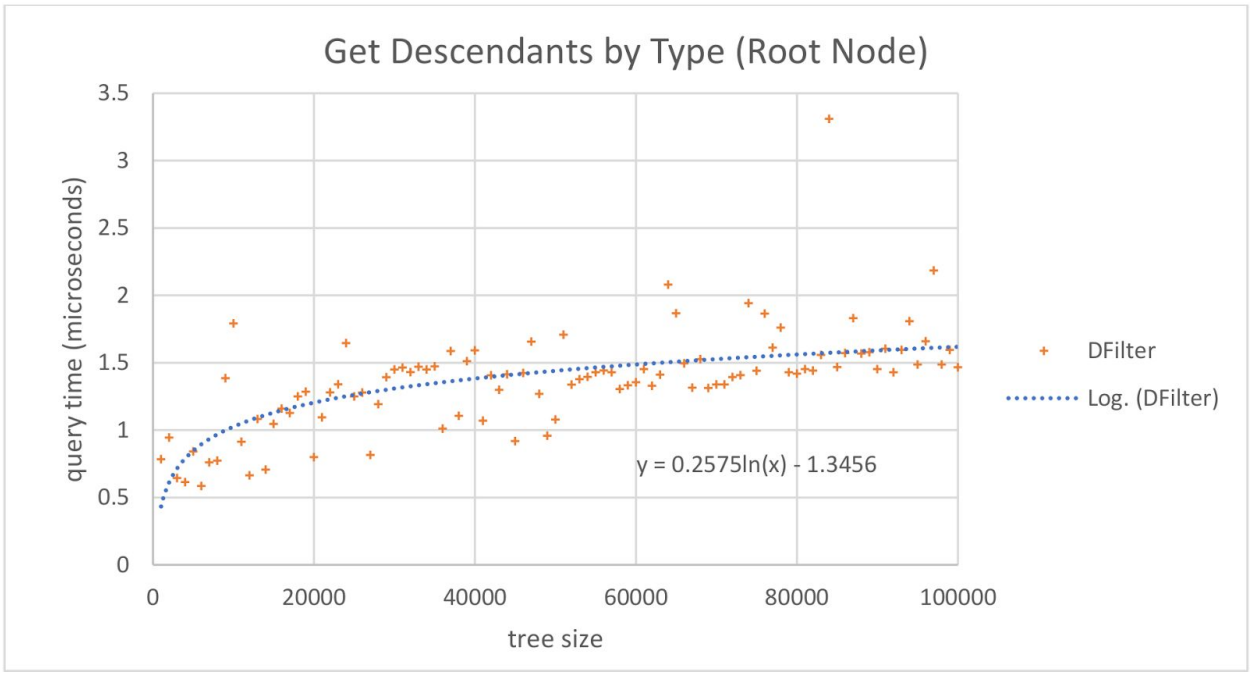
performing expensive work within the logarithmically bounded confines of the AVL tree give the embedded scheme an edge over the lazy scheme, which is at the mercy of the verticality of the tree in question. Likewise, the passive successor tracking scheme in practice seems to result in very short update travel paths to other nodes, despite the edge cases where this is supposed to be slow. For this reason, it is my recommendation that people use the embedded-passive variation of the DFilter algorithm.

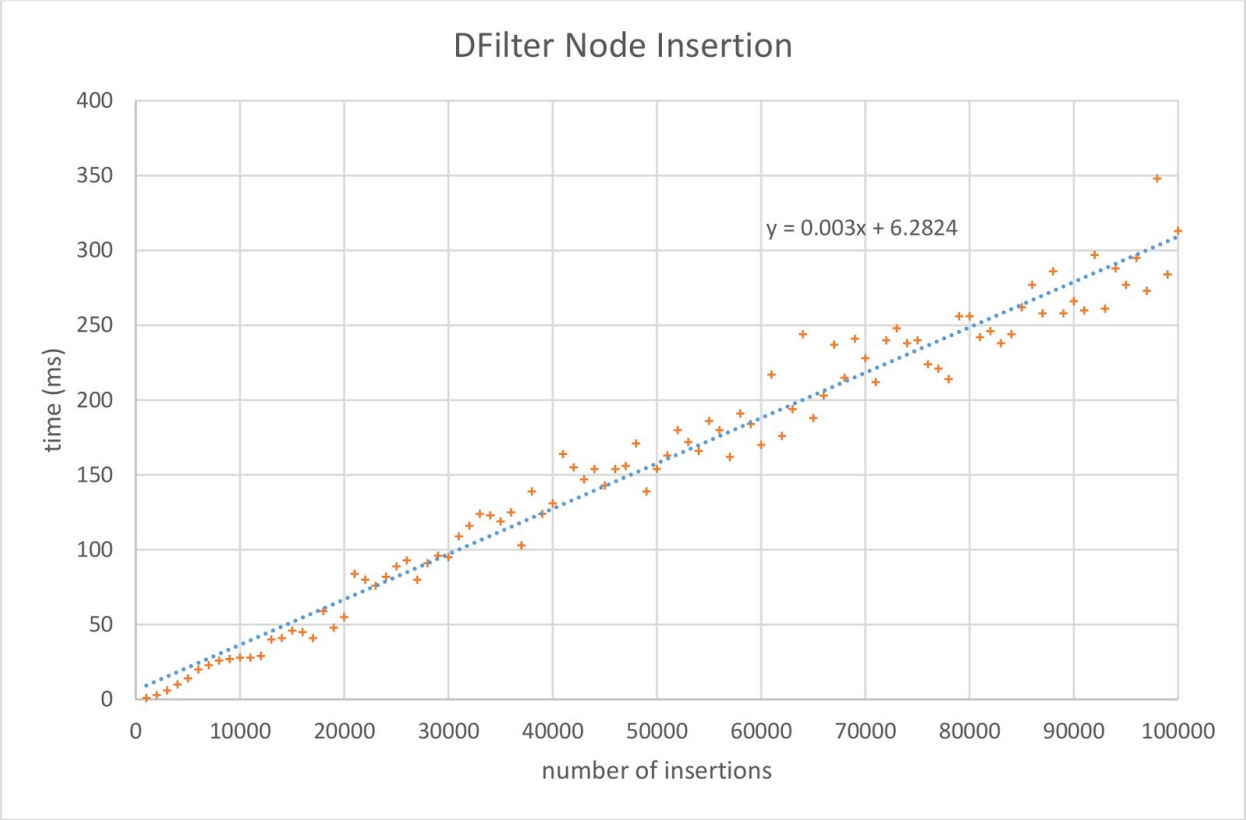
Below, we compare the average performance of the four DFilter approaches (averaged together) versus the naive approach, which again is simply traversing all descendants until the set of descendents that are of the desired type are found. I also provide a log scale so the DFilter data can be visible, as it hugs the x-axis. As you can see from the graph, DFilter is constant time compared to the naive approach (this is true regardless of which algorithm variant is used).

After this, I also provide a graph displaying the average amount of time each of the four algorithms (averaged together) takes to build a tree of size n node-by-node. A linear best-fit line is provided, meaning that node insertions using DFilter are linear in the number of insertions being performed, or rather, a single insertion is constant time in practice.

Thus, we have accomplished what we set out to do — we wanted to perform get-descendents-by-type queries in near-constant time, and perform updates to the underlying indexing data structure also in near-constant time, and on both counts we were successful.


Get Descendants by Type (Root Node)
query time (microseconds)
1400
1200
1000
800
600
400
200
0
0
20000
40000
60000
80000
100000
tree size
Dfilter
Naive


Get Descendants by Type (Root Node)
query time (microseconds)
3.5
3
2.5
2
1.5
1
0.5
0
0
20000
40000
60000
80000
100000
tree size
y = 0.2575ln(x) - 1.3456
DFilter
Log. (DFilter)



Works Cited

[1] Kelly, Samuel Livingston, "AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem" (2014). Dickinson College Honors Theses. Paper 147. http://scholar.dickinson.edu/student_honors/147

[2] Kelly, Sam, “Hierarch: Empirical Study” (2016). Brown University. Software Engineering. Online. https://github.com/sam0x17/hierarch/blob/master/empirical_study

[3] J. Lu, T. W. Li, C.-Y. Chan and T. Chen, "From region encoding to extended dewey: On efficient processing of XML twig pattern matching.," in *Proceedings of the 31st International Conference on very large Databases*, 2005.

[4] D. Quinlan, M. Schordan and J. Too, "ROSE Compiler Framework," *Lawrence Livermore National Laboratory*, 2015. [Online]. Available: http://rosecompiler.org.