Fast Type-based Querying & Indexing of Dynamic Hierarchical Data

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

1. Introduction

In this paper we describe a novel indexing scheme and search algorithm for typed, in-memory trees that is robust to deep structural tree 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 throughout the tree. We then benchmark our algorithm against the naïve tree traversal approach commonly used to answer these queries in static analysis and similar domains. 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. We 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 us the ability to rapidly find the number of nodes (or return an iterator over these nodes) that are descendants of the specified tree node.

1.1 Motivation

Efficient, real-time analysis and manipulation of typed, dynamic, hierarchical data is becoming increasingly important across a variety of previously unrelated domains, including XML indexing [1] [2] [3], static analysis [4] (of source code), Document Object Model (DOM) manipulation, online social networks and encyclopedias, search engines, file systems, and even data science. In tree-based datasets where there is a meaningful notion of node "type", such as abstract syntax trees (ASTs) or HTML/XML files, type-based queries that access descendants of some node by type (e.g. "find all IMG tag nodes within this particular DIV node") are common and can be seen as a basic building block for building more complex queries that look for specific paths or so-called "twig" patterns that are commonly optimized in existing work. Accessing descendant nodes by type is an intuitive and popular way of navigating through typed trees, and is the basis of powerful technologies such as JQuery DOM selectors which have revolutionized the way client side developers interact with the DOM structures.

Previously, indexing and optimizing for these types of queries was done mainly in static domains where deep structural tree changes (e.g. adding or removing an inner or root node) are not permitted or are uncommon [4] [1]. With completely static trees, it is possible to exploit pre-order indices to answer these and a variety of other powerful containment-based queries in constant time [4]. Unfortunately, when using these and similar indexing schemes, even simple structural changes to the tree in question will typically require an inefficient $O(n)$ re-indexing operation [4], which makes these schemes impractical for *dynamic* trees. While in some cases efficient indexing algorithms do exist for these types queries in the form of twig-pattern matching on modern XDBMSs, these systems and the algorithms they rely on are designed for massive trees on the order of many gigabytes that do not fit in system memory, and would perform poorly (worse than the naïve tree traversal approach) with smaller trees that fit entirely in system memory, such as the abstract syntax trees encountered in static analysis and compilers, and the DOM in modern web browsers.

Supporting structural changes is important for indexing algorithms in this space. Dynamic programming language implementations deal with syntax trees that constantly change during runtime, and web browsers must manage and efficiently query over dynamically changing HTML documents that can have hundreds of thousands of nodes and hundreds of instantiated tags (node types). Even XDBMSs must sometimes deal with frequently changing XML files.

In domains such as static analysis and compilers, and even in mission-critical applications such as memory leak detectors and malware scanners, the brute-force approach of performing many depth-first tree traversals to search through an AST for descendants that are of specific types is extremely common [4]. In Compass, a static analysis suite developed by Lawrence Livermore National Laboratory as part of the ROSE compiler framework, software flaw scanners (“checkers”) are itemized as individual depth first traversals that could be described logically as a series of nested get descendants by type queries [4] [5]. Each checker must perform one or more depth-first traversals, even though in an information-theoretic sense, this should only need to happen once given a proper indexing scheme regardless of how many checkers need to run. Despite the fact that performance is extremely important in many applications of static analysis, as of 2015, there are no existing publications that address this issue of avoiding repeated tree traversals when finding descendants by type. Given the concerning popularity of inefficient tree traversals as the preferred way for finding all the descendants of a node that are of a specific type [5], and the lack of existing indexing schemes particularly suited for this purpose, the need for new algorithms and indexing schemes in this space is quite clear.

Specifically, there is a need for a general purpose, in-memory, typed tree indexing scheme that can answer get descendants by type queries in near-constant time (i.e. return a count and/or a constant time iterator), but in such a way that tree modifications like insertion or deletion of an inner node are inexpensive and do not corrupt or alter existing portions of the index.

1.2 Contributions

We have identified a sizeable gap in the existing research on typed tree indexing and we have attempted to fill this gap with a novel algorithm and indexing scheme with proven performance and numerous real-world applications. Our indexing scheme delivers near constant-time get descendants by type query results on in-memory, dynamic trees where previously this was typically done in $O(n)$ using tree traversals. We have not only created a strong opportunity for the static analysis, programming languages, XML, and web browser communities to navigate and query over typed trees more efficiently, but in doing so we have also defined a novel data structure (DFI filters) which makes pre-order index based containment detection techniques robust against dynamic updates that previously made these methods intractable.

1.3 Outline

First, we review some related and relevant work in the XML indexing, static analysis, and database communities as well as some background material (namely, pre-order index based containment). Using insights based on AVL trees and the containment-preserving properties of pre-order indices, we build towards and eventually describe our novel data structure, the DFI filter. Next we unveil our search algorithm and indexing scheme by extending a basic DFI filter to support multiple node types (we refer to this extended version as a "DFilter") and derive the time and memory complexity of its basic operations including the time complexity of get descendants by type and insertion. At this point, we take a moment to briefly describe our open source C++ implementation of the DFilter algorithm (dubbed "Hierarch"), including some of the development hurdles we ran into and some of the tradeoffs that exist between different variations of the algorithm. In the section that follows, we discuss our experimental benchmarking setup and present the results of our benchmarks. In the final section, we summarize our findings and discuss some potential opportunities for future work in this space. Finally, in the appendix, we include graphs of our experimental findings and a link to the GitHub source repository for Hierarch.

2.1 Related Work

There are surprisingly few indexing schemes in existing published works that address hierarchical data to begin with, and of those that do, most are only tangentially related to what we are trying to accomplish here with our DFilter search algorithm and indexing scheme because of our focus on in-memory trees and tree modifications.

Lu et al. [1] describe a revolutionary node encoding scheme that captures the entire root to current node path of a node in a simple label that is compact in memory and can be used with standard string searching algorithms (including wildcards) to search for complicated twig-patterns. They use this encoding scheme to derive what has since become the core twig-pattern matching algorithm in the XML community. Since this approach encodes entire paths into each node, it could not meet the node insertion/deletion performance we are looking for because they do not describe an update process that could propagate a deep internal tree change to the rest of the tree efficiently (i.e. if a parent is added to the root node, all node labels much change resulting in an $O(n)$ update operation). Of particular use to us, however, is their discussion of older region encoding methods, including a rudimentary version of the pre-order index region encoding scheme used in this paper. They describe pre-order indexing as intractable specifically because of the index change propagation problem we address and overcome in this paper.

In our previous 2014 paper, Kelly [4] describes a novel indexing scheme that answers get descendants by type queries in constant time with a roughly $O(n)$ index generation procedure. This work was largely the inspiration and starting point for the DFI filter data structure derived in this paper. The problem with this older approach is that the way the index is designed, it fundamentally cannot be updated without a linear update scan over all subsequent nodes in the tree. Because it is based on pre-order indices, when one index changes, all subsequent indices must be incremented by one. Otherwise, this approach is optimal so long as the tree being indexed can be expected to remain static, since constant time query speeds are obtained in the paper.

Researchers at Microsoft [6] have developed a powerful hierarchy-based datatype (HierarchyID) into SQL Server 2008 that is capable of representing and manipulating typed hierarchies, and includes the ability to perform get descendants by type queries among others. At the time of writing, Microsoft has not disclosed the indexing scheme (if any) that is employed to optimize these types of queries, but it might be fruitful to explore HierarchyID further and benchmark against it in the future.

The research team behind VTD-XML [7] has spent the last ten years developing a powerful, general purpose XDBMS (XML database management system) well-suited for large scale, multi-server XML databases. The tag-based node labeling scheme they employ is intriguing, and as with HierarchyID, it might prove fruitful to benchmark against VTD-XML as a part of future work, especially if Hierarch is eventually developed into a more mature database system.

2.2 Background (DFI Filters)

Pre-order depth-first index labeling is extremely powerful in that this labeling scheme encodes containment information about a node's descendants within a simple integer. Specifically, a crucial property the algorithms in this paper rely on is the fact that given a tree where each node has been annotated with its corresponding pre-order depth-first index, if you select some node A, all descendants of A will have pre-order indices greater than the pre-order index of A, and *less* than the pre-order index of the next node (B) that would be visited in a pre-order traversal once A's descendants have been completely explored. For simplicity, we refer to this node B as A's "successor" (or in the code, "post-order successor"). Given A's index, and B's index, one can derive in constant time the number of descendants of A simply by subtracting these indices. Not all nodes in a tree will have such a successor if they are towards the end of the tree. These nodes are even easier to deal with since their successor index is equal to the size of the tree.

The problem with using pre-order indices to store containment information is that maintaining these indices is extremely difficult, and finding a node's successor isn't always easy. Consider node A from our previous example. Suppose that we add a child to A. To annotate this child, we must assign it a pre-order index, however to avoid duplicates, we must increment *every node in the tree* after our new node in order to free up an unused pre-order index. To avoid this predicament, we could design an update scheme that allows us to only change a few nodes each time we add or subtract a node from our tree. By adding modification IDs and a cached version of each node's parent's pre-order index to each node, and ensuring that the root node always has the most up-to-date modification, we can propagate changes up the tree such that later when other nodes try to figure out what their pre-order index is, they traverse up the tree until they find an up-to-date parent. Then by observing the difference between the child's cached parent pre-order index, and the parent's new pre-order index, the modification that was enacted on the parent (i.e. an index shift of +1) can be detected and propagated back down to the requesting node. This technique requires that every node have parent links, and will work even if numerous tree updates occur

without having to update descendants and neighboring nodes explicitly. The worst case time complexity for this update process is $O(H)$ where H is the height of the tree in question, but it is often much better than this in practice since usually most nodes close to the root are already up-to-date. This is much better than the $O(n)$ process that would be needed to manually change each subsequent pre-order index on a node insertion.

With this setup, we are still limited by the topology of whatever tree we are working with. Ideally, an indexing algorithm will have the same performance regardless of the topology of the tree that has been indexed. To achieve this, we can create an AVL tree where the key is each node's pre-order index. The AVL tree allows us to find nodes based on their pre-order index in $O(\log n)$ time, but how can we keep an AVL tree updated when insertions/deletions happen in the original tree without corrupting our pre-order indices? The answer is to create two-way links between each normal tree node and its corresponding AVL tree node. Now instead of statically sorting each AVL tree node based on a stale pre-order index, we can define our AVL tree's compare operation such that it reaches out to the normal tree node and performs the update process we outlined above in the event that the node's pre-order index is out-of-date.

The same caching/modification ID scheme outlined above can be used to track updates to each node's successor. This is aided by the fact that most of the time, it is quite easy to find a node's successor. It turns out that it is sufficient just to check if there has been a pre-order index change (either a node insertion or a node deletion) *between* a node and its presumably out-of-date successor, and only recalculate the successor (via a short exploration of the nodes immediately to the right and possibly above our source node) when changes occur in this between region. The reason for this is simple: if a pre-order index change occurs before or after both our node and our node's successor, than both nodes will simply be incremented or decremented by 1, or not at all, which will preserve the correctness of their node, successor relationship. It is only when one node has its index change, and the other does not, that we run the risk of corrupting our successor information.

By creating two-way links between AVL nodes and our tree nodes, and by employing an update propagation technique, we can store successor links, modify the underlying tree, incorporate updates when there is a tree change, get the up-to-date pre-order index of any arbitrary node in roughly constant time (amortized with tree updates), and find a particular node (via the AVL side of the tree) based solely on its pre-order index.

It also turns out we can do even better when it comes to propagating pre-order index changes, but at the price of a very difficult implementation. If, instead of propagating up the normal tree, we add rhs_offset and lhs_offset attributes to each AVL node, we can follow the properties of binary search trees (namely that everything to the left is less than the node in question, and everything to the right is greater than the node in question) to achieve roughly the same result but instead of $O(H)$ this operation now only costs $O(\log n)$. The problem is when AVL rotations kick in, it is very easy for a rotation to corrupt this propagation resulting in nodes being assigned incorrect pre-order indices. Each rotation type must be manually accounted for and changed so that the modification IDs and offsets are not distributed.

Regardless of which way one chooses for maintaining pre-order index propagation, the result is a DFI Filter – a novel data structure for quickly querying over a 1-type tree and cheaply tracking updates to the pre-order indices of all contained nodes with very little overhead.

3. DFilter Indexing Algorithm

Using the insights we used to develop our simple DFI Filter, we can extend this structure so that it supports get descendants by type queries. If for each node type in the tree, we create an AVL tree containing only the nodes of that type in the tree (along with our usual two-way link back to the corresponding main tree node), where the key is the pre-order index of the node in the main tree, and if we also store and maintain on each "type" AVL node (using the same propagation technique described in the previous section) a "local" pre-order index that only applies to nodes of that type, then we can use pre-order indices in the main tree as an index *into* these smaller type AVL trees to quickly find the start and end node of a get descendants by type query. A search for a pre-order index in a type AVL tree will result in the closest node the tree can offer to the specified main tree pre-order index.

Given a node in the main tree, we can use our link to its successor to immediately find both the node's preorder-index, and the pre-order index of its successor. With this information, we can then perform two searches in the desired type AVL tree for the pre-order indices of the node and its successor. If two nodes are returned both with pre-order indices between the node's pre-order index and the node's successor's pre-order index, then these are precisely the first and last node of the specified type in the list of descendants of the node we are inspecting.

Using DFilter, get descendants by type can be solved in $O(\log n_t)$ where n_t is the number of nodes of the search type that are contained in the overall tree. Given a new, unindexed tree, DFilter can index said

tree in $O(n)$ using a modified depth-first traversal. Once an index is initialized, insertions either cost $O(\log n)$ or $O(H)$ depending on which of the two pre-order index propagation schemes was used.

4. Implementation Details

We implemented the basic insertion and get descendants by type methods of the DFilter algorithm in a C++ library called Hierarch. For AVL trees, we integrated with the GNU LibAvl library and used their implementation of PAVL (AVL tree with parent pointers) for our base AVL tree and type AVL trees.

We fully implemented the two basic key operations, namely insert (inserts a new tree node into the specified position in the tree and updates the tree and underlying index accordingly), and get descendants by type. We decided to omit the destructive operators such as replace and remove since propagation for these is implemented no differently than it was for insertion. For a pre-order index propagation scheme, we favored the simpler version that drives updates directly up to the root of the main tree because this was significantly easier to implement and test, although a stale branch remains in the Hierarch GitHub repository with the

Our implementation is structured into a DFilter header file and cpp file, a test.cpp file with a number of functions designed to fully test every aspect of the insert and get descendants by type methods, and is automatically compiled and linked against our modified version of the GNU LibAVL library upon running the build.sh script provided with the implementation. We left most methods public to enable easy tweaking and research-oriented use of the code base. We also took the time to provide a special iterator that is returned by get descendants by type that uses a constant time algorithm to increment to the next node in the result.

We took great care in the design of the Hierarch code base to properly test every promised property of our algorithm, including a deep testing suite that continuously verifies that every node in DFilter is correct while insertions and queries are carried out. This process is a bit slow, but the result is we can be extremely confident in the accuracy of our implementation. For generating random trees, we consulted experts at LLNL for statistics on real-world average branching factor and node type distribution of nodes in real-world ASTs (since these are a good model for us to use), and we used these settings as closely as possible in our random tree generation code.

5.1 Experimental Design

Our main goal with our experiment was to assess the performance of our indexing scheme both when it comes to updating the index with new information (i.e. added or removed nodes), and executing queries. We consulted compilers industry experts to generate as close to real-world-AST as possible tree topologies, so we can make the claim that the trees we generate are effectively "real-world" trees.

The benchmark suite runs automatically after the testing phase of the main hierarch executable. For get descendants by type, we benchmarked against the naïve depth-first search tree traversal approach to finding descendants by type. In our case, this consisted of a simple stack-based depth-first visitor that used an int variable to keep count of the nodes it has seen of the desired search type. Because both algorithms return almost instantly on trivial starting nodes that have few or no descendants, we pruned these from the benchmark and only allowed nodes with at least 20 descendants. We ran two separate tests for get descendants by type – one on random nodes with at least 20 descendants, and one focused entirely on the root node since typically this is the most difficult node to perform queries on since it has the most descendants. We also ran a quick benchmark of the insertion method. In this benchmark, we had the DFilter continuously add a new random node to the main tree building it up from size 1 to size 100,000.

We performed our benchmarks in an Ubuntu 14.04 virtual machine running in VirtualBox on Microsoft Windows 10 on a Lenovo X230 laptop with 8 GB of RAM (4 GB were allocated to the VM, and the VM was given access to all CPU cores).

5.2 Results

(see appendix for graphs)

Our strongest result was that calling get descendants by type runs in ~1.5 microseconds even on trees with 100,000+ nodes. On the root node test, get descendants flat-lined compared to the slow progress of the manual depth-first tree traversals. This makes sense, since root-level queries should be most difficult given that they have the highest number of descendants. We also graphed the root level get descendants by type data by itself and found that it had a logarithmic trend-line, which makes sense given that we expected it to have $O(\log n_t)$ performance.

Most interesting were the results for the random node get descendants by type benchmark. The naïve algorithm exhibited erratic behavior, but with increasing maxima as you move across to larger tree sizes. DFilter was much more performant on average, and was not observably affected by the increase in overall tree size. This is likely because DFilter is much more dependent on the number of nodes of the type being searched for, than the overall size of the tree, since the only AVL search performed is done within the type AVL tree. This fact gives DFilters the odd property that the more types you add, the faster the algorithm will be able to hone in on results.

In addition to benchmarking get descendants by type, it was also important to conduct thorough benchmarking of the insertion method we provided in Hierarch. We found that 100,000 insertions can be conducted in roughly 313 milliseconds. Our graph and trend-line revealed that the insertion method exhibits slightly linear behavior, but with an extremely gradual and forgivingly low slope. This was expected, since we knew insertion to be $O(H_t)$ where H_t is the height of the type AVL tree being searched, and it is quite obvious that $H_t \ll n$.

6.1 Discussion

In the course of this paper we have presented a novel data structure (the DFI Filter), and built this data structure up into our novel dynamic in-memory typed tree indexing scheme and searching algorithm, DFilter. We have derived the theoretical underpinnings of DFilter and DFI Filters, namely the containment-wielding properties of pre-order depth first indices, and solved the old problem of updating pre-order indices on a live tree. We have furthermore shown that DFilter is extremely competitive at integrating tree updates and answering get descendants by type queries, with $O(\log n_t)$ and $O(H_t)$ time complexities for the get descendants by type and insertion methods respectively.

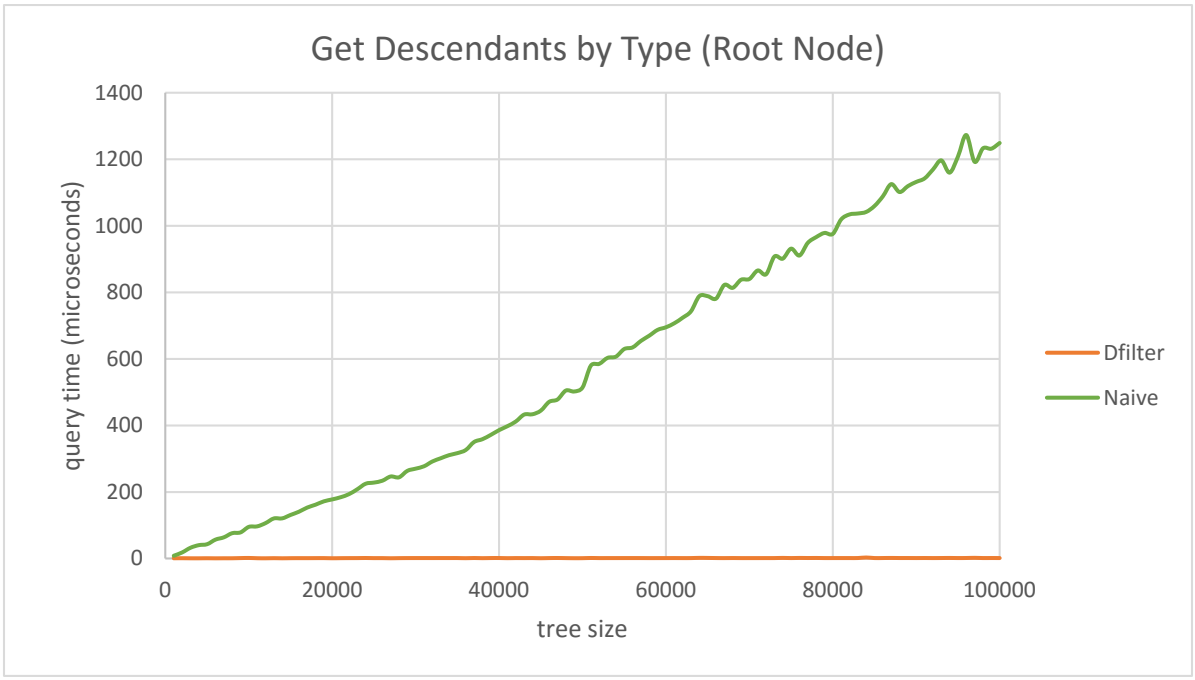
Through careful analysis and benchmarking, we have shown that the DFilter approach is superior across the board to brute force pre-order depth first traversals aimed at finding the descendants of a node that are of a particular type, and we have affirmed the theoretical time complexities of these operations using trend-lines.

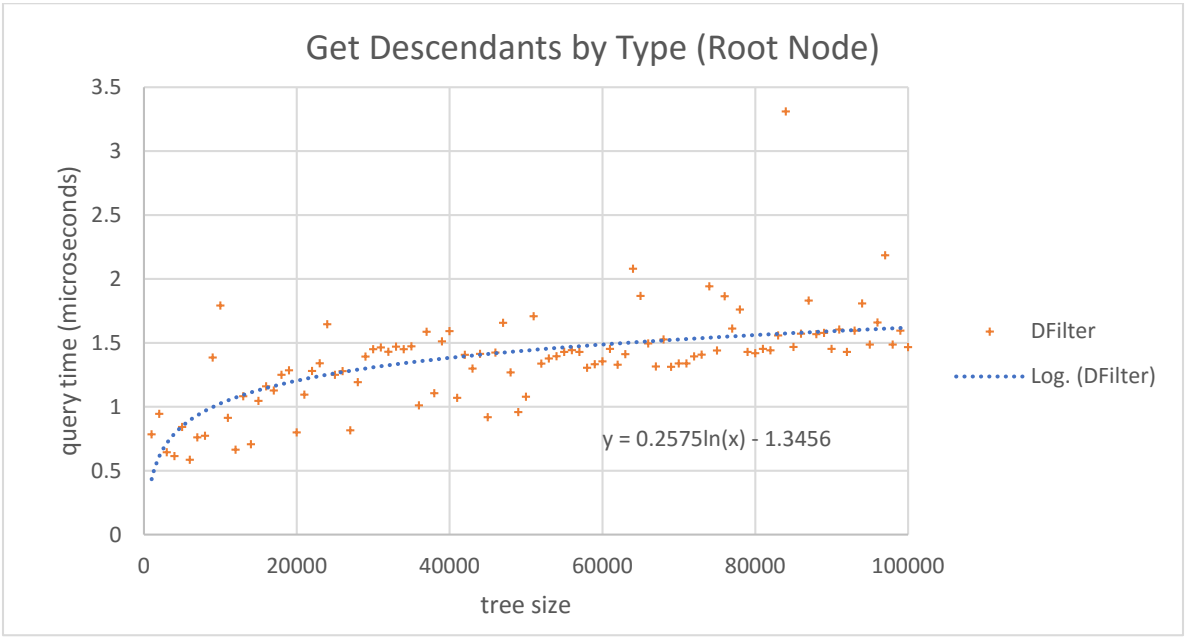
6.2 Future Work

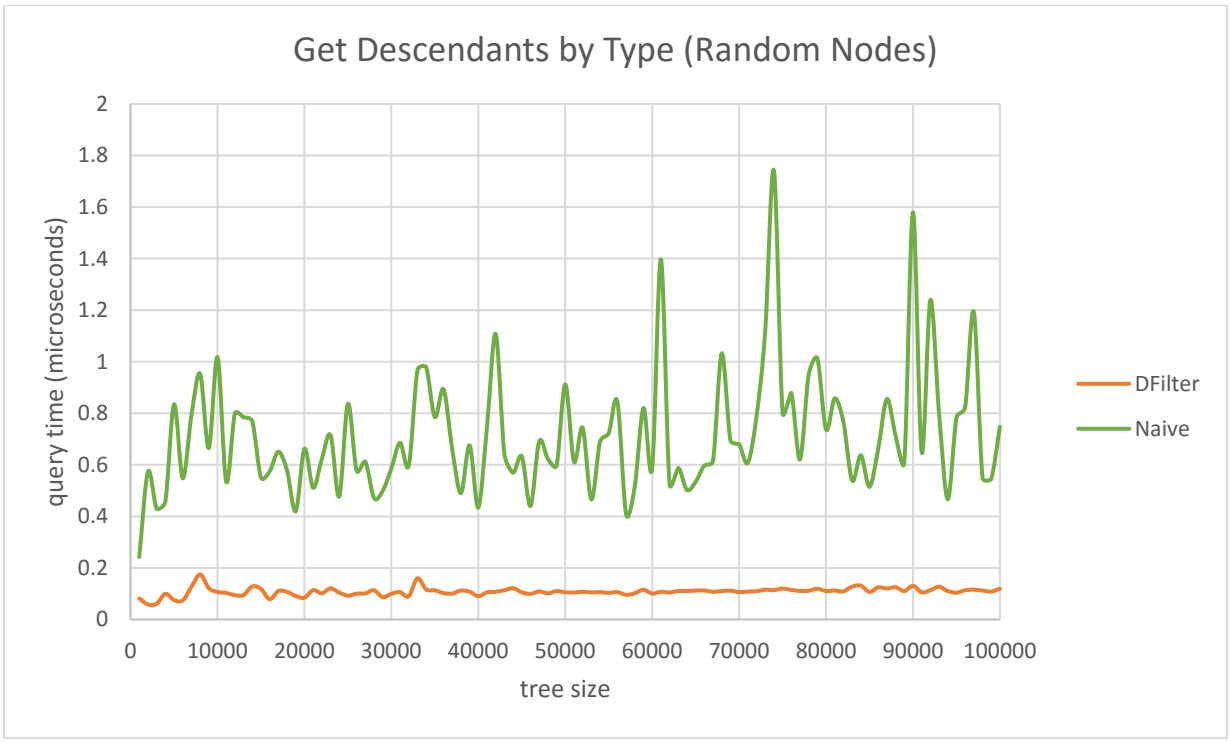
There are ample opportunities to conduct additional research on the DFilter algorithm and the related DFI Filter data structure. This new technology could potentially have powerful applications in static analysis (where it was originally intended to be used), file systems, and even databases.

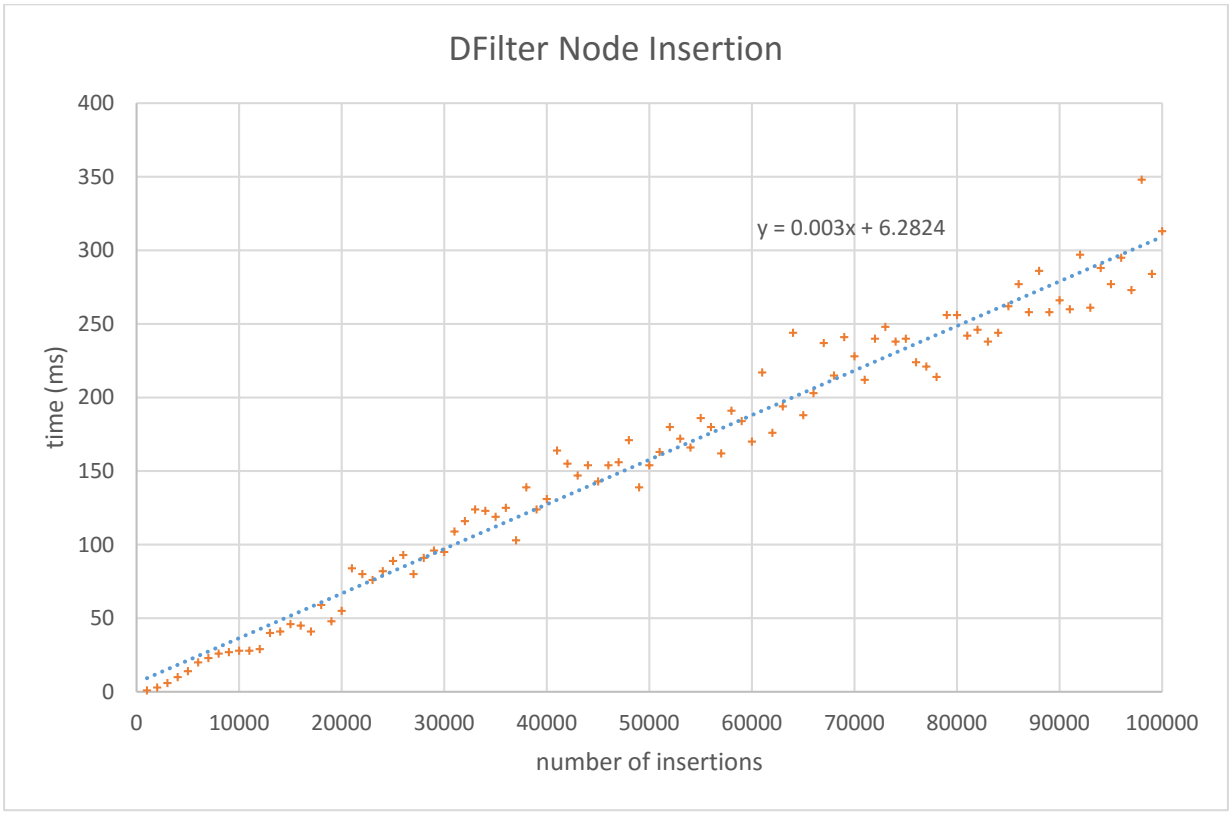
It would be desirable to perform benchmarks against the best the XML indexing community has to offer, in addition to Microsoft's elusive HierarchyID. There is also much room for improving and expanding upon the code for Hierarch, which could potentially be developed into a more general system that supports a variety of queries and indexes over hierarchical data.

Appendix






Get Descendants by Type (Random Nodes)
query time (microseconds)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
tree size
DFilter
Naive


DFilter Node Insertion
time (ms)
400
350
300
250
200
150
100
50
0
y = 0.003x + 6.2824
0
10000
20000
30000
40000
50000
60000
70000
80000
90000
100000
number of insertions

GitHub

github.com/samkelly/hierarch

Works Cited

[1] 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 data bases*, 2005.

[2] W. Alink, R. Bhoedjang, P. A. Boncz and A. P. de Vries, "XIRAF-XML-based indexing and querying for digital forensics," *Digital Investigation 3,* vol. 3, pp. 50-58, 2006.

[3] L. Chen, C. Liu, Q. Liu and K. Deng, Database Systems for Advanced Applications, Brisbane: DASFAA 2009 International Workshops: BenchmarX, MCIS, WDPP, PPDA, MBC, PhD, 2009.

[4] S. Kelly, *AST Indexing: A Near-Constant Time Solution to the Get-Descendants-by-Type Problem,* 2014.

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

[6] I. Ben-Gan, L. Kollar, D. Sarka and S. Kass, Inside Microsoft SQL Server 2008 T-SQL Querying: T-SQL Querying, Microsoft Press, 2009.

[7] "VTD-XML: The Future of XML Processing," SourceForge, [Online]. Available: http://vtd-xml.sourceforge.net/technical/0.html. [Accessed 18 November 2015].