Measuring the Popularity of Get-Descendants-by-Type Queries in the Wild

CSCI 2340 Final Paper (Empirical Study)

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

ABSTRACT

We conduct an empirical study to measure the popularity of get-descendants-by-type (GDBT) queries over dynamic, typed hierarchical data in the wild, finding that over 56% of jQuery selector calls can be expressed *strictly* as a pure composition of GDBT queries. We argue that this data strongly supports our hypothesis that in practice, the majority of queries made against typed trees can be expressed entirely as a composition of GDBT queries, and our high-level belief that GDBT queries are the basic building block of non-trivial hierarchical data analysis and exploration. By finding strong evidence for this hypothesis, our study also suggests that further research involving typed trees should single out and specifically focus on optimizing GDBT queries, and validates our ongoing research into GDBT query optimization via novel indexing algorithms and data structures.

Introduction

In this study we attempt to measure the popularity of get-descendants-by-type (GDBT) queries over dynamic, typed hierarchal data in the wild. Despite the existing wealth of research and publications in XPath, XML indexing, compilers, static analysis, and even areas such as databases and file systems where typed trees are also important, to date no previous research other than our prior paper [1] has specifically addressed GDBT queries or singled them out as special among the potential queries and search operators that can be executed over typed trees. Our goal is to provide a concise, cross-domain definition of GDBT queries, to conduct an empirical study that applies this definition to a large dataset of existing source code that operates largely on dynamic, typed hierarchical data, and finally to use this data to measure the overall popularity of GDBT queries in the wild.

Motivation

Our experience in a number of domains where queries over typed trees are commonplace (including static analysis, software vulnerability detection, databases, file systems, and DOM manipulation) has culminated in our belief that GDBT queries are fundamental to non-trivial hierarchical data analysis and deserve special treatment and optimizations. This belief has motivated our ongoing research project Hierarch/DFilter [2], which specifically optimizes GDBT queries using novel indexing algorithms and data structures.

Conducting this empirical study allows us to gauge the relative impact our Hierarch/DFilter research will have on the computer science community as a whole. Since the development of Hierarch/DFilter was motivated by our strong belief in the ubiquity of GDBT queries, our hypothesis is that in practice, the majority of hierarchical data queries are either GDBT queries or are purely composed of GDBT queries. If this hypothesis is true, it strongly implies that our Hierarch/DFilter research will have a positive performance impact on the majority of typed hierarchical queries, and that research into typed trees should specifically focus on and optimize GDBT queries. In other words, if GDBT queries are truly as commonplace as we predict, then our ongoing research could have a groundbreaking effect in many domains where hierarchical data analysis, exploration, and manipulation is performed.

Background

Solving the get-descendants-by-type (GDBT) problem involves answering typed-tree queries 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. For example, within the Document Object Model (DOM) of a web page, a selector that searches for all IMG tags that are contained within a specific DIV tag would count as a GDBT query, where the DIV tag is A, and $T = IMG$.

Anecdotally, at least, GDBT queries seem to be ubiquitous, appearing in a wide variety of domains and sub-fields of computer science [1][2], including databases, filesystems, compilers, static analysis, operating systems, web browsers, artificial intelligence—essentially anywhere that typed trees are regularly constructed, traversed, or manipulated in some way.

In this paper we also introduce the notion of “pure” and “non-pure” GDBT queries. We define “pure” GDBT queries as hierarchical queries that could be represented *purely* as a composition of classic GDBT queries, and “non-pure queries as queries that contain at least one non-trivial (requires traversing more than one additional tree node), non-GDBT portion. For example, a query that searches for all IMG elements that are contained within DIV elements in a web page DOM would qualify as a pure GDBT query, since it can be represented by a nesting/composition of “find all DIV tags under the root node” and “find all IMG tags under node A”. An example of a “non-pure” GDBT query, on the other hand, would be a query that searches for all DIV nodes that have more than 20 children. While this query contains a GDBT query (find all DIV nodes that descend from the root), the “more than 20 children” portion cannot be broken down into constituent GDBT calls, as it is calculating something non trivial that does not involve the type system or the node hierarchy. Similarly, a simple query of the form “find all nodes with less than 10 children” contains no GDBT portions, and is therefore also non-pure, by definition.

Existing Work

Despite the apparent commonality of GDBT queries, existing work outside of our own research does not focus on these types of queries, nor does it appear to deem them as deserving of special attention or targeted optimizations.

In [1], we describe an indexing algorithm for typed, *static* trees that leverages per-type depth-first index properties to solve the “get-descendants-by-type” problem in near-constant time. While this approach solves the GDBT problem in near-constant time, a full $O(n)$ re-indexing operation is required every time there is a structural change to the tree in question, since the subtraction or addition of even a single node results in a linear number of nodes in the tree receiving new depth-first index assignments. Thus the weakness of this approach is that it can only handle static trees—dynamic updates require full re-indexing.

In the draft publication [2], we describe a newer, much more complex indexing algorithm and data structure that offers roughly the same near-constant time performance on GDBT queries as that offered by [1], but with equally performant index updates in the face of node additions and deletions. This approach is entirely tree-based, and takes advantage of the novel DFilter data structure, which materializes a type-based view into existing trees with dynamic successor tracking and by managing per-type AVL trees that track the relative depth-first indices of nodes in the tree [2]. Structural tree changes are propagated through simple inexpensive index update operations that in practice are no more demanding than a GDBT query [2].

A number of algorithms and data structures exist [3][4][5][6][7] and are discussed at length in [2] that inadvertently optimize GDBT queries, notably in XPath and XML indexing research, but none that do so as quickly as [2] or in a way such that the underlying index is robust to structural changes, as is required in dynamic domains like DOM traversal, file systems, and databases.

Hypothesis

As stated previously, this study’s overarching hypothesis is that in practice, the majority (i.e. more than 50%) of queries made against dynamic, typed trees can be expressed entirely as a composition of GDBT queries.

Methods

To test our hypothesis, it is necessary that we develop some notion of what “in practice” means when it comes to hierarchical queries, and that we use this notion to pick (or compile ourselves) one or more datasets that reflect the demographics we want to measure. In the sections that follow we discuss our process for selecting a dataset, including the considerations that resulted in our focus on jQuery selectors. We then discuss our overall experimental design, including our method for identifying GDBT queries within jQuery selectors, and a concise overview of the statistical information we generate based on our dataset.

Dataset Selection

We have identified four primary domains where hierarchical queries are regularly made against typed trees and real-world instances of these queries can be found in publicly available source code (e.g. on GitHub).

The first is the software vulnerability "checkers" from the source code of the ROSE Compass 2 project. Over 100 checkers exist, each using a custom AST-querying language [5] to traverse typed abstract syntax trees in search of particular vulnerabilities. While this would be an easy dataset to check, the fact that we have previously worked on this project would be a potential bias, and thus any conclusions we draw from this data would be potentially tainted. While many other software vulnerability checkers exist, few are open source, and those that are tend to use visitors instead of a type-based query language, making these more difficult to analyze, and otherwise sparse and undesirable as a target dataset.

Another domain where hierarchical queries are common is in SQL, however the mapping of SQL into hierarchical tree queries is generally problematic, since modern relational database schemas are typically not strictly hierarchical (you cannot perfectly represent the tables and indexes of a typical relational database with a single hierarchical view, as many disjoint hierarchies can and usually do exist over the same data). That said, Microsoft's SQL Server recently introduced a feature called HierarchyID [6], which is queryable via a syntax resembling XPath and allows for the creation and querying over what are essentially typed trees. Unfortunately very few projects have adopted this syntax (it is not very widely advertised), so collecting a large set of source code examples of HierarchyID queries would be virtually impossible at present.

Likewise, XPath is very common as a query language for posing hierarchical queries against typed trees, with XPath libraries and implementations in virtually every programming language and environment. Unfortunately, we found that while XPath support is extremely common, XPath *usage* is largely fragmented between the scores of XPath implementations, and is not commonplace in any particular language or framework that has large amounts of source code available on GitHub or other open-source repository sites. Thus XPath is too rare for us to be able to compile a reliable dataset of hierarchical queries.

Where else can we look for hierarchical queries being posed against typed trees frequently, and in open source code? Perhaps the best answer is in jQuery selectors. The Document Object Model (DOM) of a web page is nothing more than a typed, dynamic tree, and jQuery's primary use is to manipulate and traverse the DOM by combining a powerful node selection engine with a framework of common operations that can be applied to groups of nodes. Indeed, it is estimated that roughly 70% of web pages overall and 96.4% of web pages that use JavaScript use jQuery in some way [8]. Thus we decided to conduct our study on jQuery selectors in the hope that these results would generalize to other domains, since other potential datasets would be too small to be convincing or too difficult to accumulate. While there are no existing studies on the distribution or commonality of jQuery selectors, a great way to test our hypothesis would be to measure the percentage of jQuery selectors that represent pure GDBT queries in the JavaScript source code available on GitHub.

Conveniently, the highly cited JavaScript 150k dataset [9] contains approximately 150,000 unique, non-obfuscated JavaScript files from the top 1,000 most popular JavaScript projects on GitHub, of which we found 138,502 files that were completely parsable, representing 64,050,938 lines of valid JavaScript source code. Because of the availability, size, and quality of this dataset, we elected to use JavaScript 150k in our study.

Experimental Design

Having selected the JavaScript 150k dataset and jQuery selectors as the target of our study, we began designing our actual experiment. Due to its ease of use and powerful string manipulation capabilities, we decided to code up our experiment using the Ruby programming language. Our overall high level process for each JavaScript file is as follows:

First we attempt to parse the .js file using the “rkelly_remix” JavaScript parsing Ruby gem. This results in either a parse error or an abstract syntax tree (AST) of the JavaScript file. We found that a negligible 3,891 (2.733%) of the files contained errors or were otherwise unparsable, and these files were excluded from our study.

Once we have an AST for a valid .js file, we identify all the potential jQuery selectors in the file by running an AST visitor that finds all function calls with function names that match “.find(..)”, “jQuery(..)”, or “$(..)”, which represent the primary APIs by which jQuery selector queries are typically made (the mapping of the dollar sign to jQuery is practically universal in modern web development). At this point we know that the “.find(..)” results are probably jQuery selectors, however the other two types could be either selectors or element wrappers (e.g. “$(elem)” where elem is a DOM node).

Most jQuery selectors are explicit, such as “$(‘div’)”, however many also use JavaScript variables or a concatenation of variables and string literals. With static analysis alone, there is little we can do to handle the cases where there is only a single variable wrapped in parenthesis, as this could just as easily be “$(e)” where e is a DOM node variable (not a selector query) as it could be “$(q)” where q is a JavaScript variable containing a selector query string. Thus we adopt the following policy: if the contents of the parentheses contain no single or double quotation characters, we assume the jQuery call is not a selector query, since we cannot differentiate between the q and e cases without dynamic analysis. Dynamic analysis in this case would be extremely difficult since we have no insight into the calling context within any given JavaScript file, and so we do not have a standard entry-point where we can begin execution. By not counting single-variable jQuery calls as potential selector queries, we avoid the risk of overestimation due to e-style calls, but in doing so we are also discarding q-style calls, meaning that our final count of jQuery selector queries is an underestimation and less than the true number.

At this stage, we perform some string transformations that handle the concatenation of JavaScript variables with string variables, such that the concatenation of any number of valid jQuery selectors is transformed into a single string. For example, “$(‘#’ + id)” would be transformed into “$(‘#id’)”, and “$(‘img’ + ‘ ‘ + ‘.large’)” would be transformed into “$(‘img .large’)”.

Next we analyze the contents of the parenthesis for each jQuery selector call in detail using a powerful regular expression we wrote that allows only the subset of the jQuery selector language that is capable of generating pure GDBT queries. For example, “$(‘img’)” and “$(‘.large div’)” would be valid, however selectors that do something more than manipulate the type system or the hierarchy of nodes would be invalid, such as those involving the “:”, “[“, or “]”

characters. The exact regular expression used can be found in the GitHub repository linked in the appendix. We treat DOM IDs, CSS classes, and element types as three distinct, valid type sets that can be queried against in the DOM. Thus if Hierarch were used to optimize the jQuery selector engine (Sizzle), individual indexes would be managed for IDs, classes, and element types respectively.

Once we have filtered out all non-pure GDBT selector queries, we record statistics on the pure and non-pure GDBT selector queries found in the current file before moving on to the next file. Once we have processed all files in the JavaScript 150k dataset, we print out the final versions of these statistics. The main statistic that affects our hypothesis is the percentage of jQuery selector queries that meet our definition of pure. If this percent is greater than 50%, then our hypothesis is true, and if this percentage is less than 50%, then our hypothesis is false. With each metric that we collect, we also calculate the mean, median, and standard deviation where applicable. We do this so we can account for outliers and skewing effects in the data if and where they exist. This is especially important since most projects will either use jQuery frequently or not at all, resulting in a potentially high standard deviation and a likely median of 0 or 1 when it comes to per-file statistics across our various metrics.

Results

Our experiment identified 142,799 jQuery selector queries, with an average of 1.031 queries per file, a median of 0, and a standard deviation of 9.422 (again, this makes sense because most files either use no jQuery, or lots of jQuery). Of these 142,799 selector queries, we identified 80,399 (56.302%) as pure GDBT queries, with an average of 0.5805 pure GDBT queries per file, a median of 0, and a standard deviation of 6.3065. Of these 80,399 pure GDBT queries, we were able to divide these into 122,774 constituent basic GDBT queries, with an average of 1.527 basic GDBT queries per pure GDBT selector query, a median of 1, and a standard deviation of 1.24. When counted on a per-file basis, we found an average of 0.8864 basic GDBT queries per file, with a median of 0 and a standard deviation of 11.8569 (this high standard deviation once again reflects the fact that some files inevitably have *many* jQuery selectors, and many files do not even use jQuery).

The most important result above is our finding that 56.302% of the jQuery selectors analyzed were pure GDBT queries. Since this is greater than 50%, this result affirms our hypothesis that the majority of jQuery selectors in practice can be expressed purely in terms of a composition of GDBT queries. A corollary of these results is that over 58% of JavaScript files in practice would presumably see performance improvements if Hierarch/DFilter were used to optimize the Sizzle jQuery selector engine (this is because Sizzle and the underlying browser selector engine do not do indexing on anything but ID selectors, resulting in a consistent $O(n)$ execution time for most jQuery selectors).

Discussion

Overall we were pleasantly surprised by the strong result of 56.302% of jQuery selector queries being entirely composed of GDBT queries. Minimally this means that, when it comes to DOM manipulation and querying from within JavaScript, GDBT queries are extremely important, in the very least outnumbering all other types of queries. Worth noting is the fact that we do not account for “partial” GDBT queries, or rather situations where a jQuery selector contains one or more GDBT constituents, but is by definition non-pure. This is largely because we could not devise a fair system for measuring the GDBT v.s. non-GDBT contributions within these queries. Nevertheless, there are many places within non-pure jQuery selector queries that could nonetheless be optimized by a Hierarch/DFilter approach, so the true percent of jQuery selectors that use GDBT calls is likely much higher than the 56.302% we measured (and this fact supports our argument even further).

If we accept the anecdotal evidence from [1], GDBT queries are also dominant in the domain of static analysis when it comes to writing visitor pattern traversals and performing queries over ASTs. While these results do not guarantee that the same will also be true in other domains such as file systems, operating systems, databases, etc., we have identified no reason to suspect that the tree compositions or query usage patterns will be dramatically different in these domains, and we think it is reasonable to expect that our results generalize across most hierarchical querying domains that involve typed trees of some kind. These results also support our view that GDBT queries are special, even fundamental when it comes to the querying, exploration, and manipulation of typed hierarchical data, and that they should be the subject of additional research.

Future Work

One important source of future work would be a study of typed tree compositions encountered in practice, including per-domain statistics on branching factor and type distribution as they relate to tree depth and tree size. This research would allow for the creation of more accurate statistical models of real-world typed trees, enhancing our ability to generate realistic synthetic trees. Most importantly this research would further assess whether in practice real-world trees meet the “roughly balanced” definition discussed in [1] and [2].

An additional future work topic would be to somehow explore additional domains and confirm that GDBT queries are as prominent as they are in jQuery. Yet another obvious topic for future work would be to optimize and benchmark the Sizzle selector engine using Hierarch/DFilter, and measure the actual performance gains of using Hierarch over the largely naive approach employed by Sizzle.

Appendix

The code and README for reproducing this empirical study can be found at https://github.com/samkelly/hierarch/tree/master/empirical_study.

The main Hierarch/DFilter research project can be found at https://github.com/samkelly/hierarch.

Works Cited

[1] S. Kelly, "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] S. Kelly, “Fast Type-based Querying and Indexing of Dynamic Hierarchical Data”, *GitHub*, 2016. [Online]. Available: https://github.com/samkelly/hierarch.

[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] 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.

[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://vtdxml.sourceforge.net/technical/0.html. [Accessed 24 November 2016].

[8] “Usage statistics and market share of jQuery for websites,” *W3Techs Web Technology Surveys*. [Online]. Available: https://w3techs.com/technologies/details/js-jquery/all/all

[9] Raychev, V., Bielik, P., Vechev, M. and Krause, A. Learning Programs from Noisy Data. In *Proceedings of the 43nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (2016)*, POPL ’16, ACM

Acknowledgements

I would like to personally thank professors Tim Wahls and Ugur Cetintemel of Dickinson College and Brown University respectively, as well as Justin Too of Lawrence Livermore National Laboratory for their continued guidance and support of my Hierarch-related research.