XViz: A Tool for Visualizing XPath Expressions

Ben Handy and Dan Suciu

University of Washington Department of Computer Science
handyman@u.washington.edu, suciu@cs.washington.edu

Abstract. We describe a visualization tool for XPath expressions called XViz. Starting from a workload of XQueries, the tool extracts the set of all XPath expressions, and displays them together with some relationships. XViz is intended to be used by an XML database administrator in order to assist her in performing routine tasks such as database tuning, performance debugging, comparison between versions, etc. Two kinds of semantic relationships are computed and displayed by XViz, ancestor/descendant and containment. We describe an efficient, optimized algorithm to compute them.

1 Introduction

This paper describes a visualization tool for XPath expressions, called XViz. The tool starts from a workload of XQuery expressions, extracts all XPath expressions in the queries, then represents them graphically, indicating certain structural relationships between them. The goal is to show a global picture of all XPath expressions in the workload, indicating which queries use them, in what context, and how they relate to each other. The tool has been designed to scale to relatively large XQuery workloads, allowing a user to examine global structural properties, such as interesting clusters of related XPath expressions, outliers, or subtle differences between XPath expressions in two different workloads. XViz is not a graphical editor, i.e. it is not intended to create and modify queries.

The intended user of XViz is an XML database administrator, who could use it in order to perform various tasks needed to support applications with large XQuery workloads. We mention here some possible usages of XViz, without being exhaustive. One is to identify frequent common subexpressions used in the workload; such XPath expressions will be easily visible in the diagram produced by XViz because they have a long list of associated XQuery identifiers. Knowledge of the set of most frequent XPath expressions can be further used to manually select indexes, or to select efficient mappings to a relational schema. Clusters of almost identical XPath expressions can also be visually identified, giving the administrator more guidance in selecting indexes. A similar application consists of designing an efficient relational schema to store the XML data: a visual inspection of the graph produced by XViz can be used for that. Another application is to find performance bugs in the workload, e.g. redundant //'s or $*$'s. For example if both `/a/b/c` and `/a/b//c` occur in the workload then XViz will draw a line between them, showing the structural connection, and

the administrator can then examine whether the // in the second expression is indeed necessary or is a typo. Finally, XViz can be used to study the relationship between two versions of the same workload, for example resulting from two different versions of an application. The administrator can either compare the graphs produced by XViz for the two different applications, or create a single graph with XPath expressions from both workloads, and see how the XPath expressions from the two versions relate.

In some cases there exist techniques that automatize some of these tasks: for example an approach for index selection is discussed in [1], and efficient mappings to relational storage are described in [3]. However, these techniques are quite complex, and by no means universally available. A lightweight tool like XViz, in the hands of a savvy administrator, can be quite effective. More importantly, like any visualization tool, XViz allows an administrator to see interesting facts even without requiring her to describe what she is looking for.

XViz starts from a text file containing a collection of XQueries, and extracts all XPath expressions occurring in the workload. This set may contain XPath expressions that are only implicitly, not explicitly used in the workload. For example, given the query:

```
for $x in /a/b[@c=3],
    $y in $x/d
. . .
```

XViz will display two XPath expressions: both `/a/b[@c=3]` (for `$x`) and `a/b[@c=3]/d` (for `$y`).

Next, XViz establishes two kinds of interesting relationships between the XPath expressions. The first is the *ancestor relationship*, checking whether the nodes returned by the first expression are ancestors of nodes returned by the second expression. The second is the *containment relationship*: this checks whether the answer set of one expression contains that for the other expression. Both relationships are defined semantically, not syntactically; for example XViz will determine that `/a//b[c//@d=3][@e=5]` is an ancestor of[1] `a/b[@e=5][@f=7][c/@d=3]/g/h`, even though they are syntactically rather different.

Finally, the graph thus computed is output in a dot file then passed to GraphViz, the graph visualization tool[2]. When the resulting graphs are large, they can easily clutter the screen. To avoid this, XViz provides a number of options for the user to specify what amount of detail to include.

The core of XViz is the module computing the relationships between XPath expressions, and we have put a lot of effort into making it as complete and efficient as possible. Recent theoretical work has established that checking containment of XPath expression is computationally hard, even for relatively simple fragments. As we show here, these hardness results also extend to the ancestor/descendant relationship. For example, checking for containment is co-NP

[1] We will define the ancestor/descendant relationship formally in Sec. 4.

[2] GraphViz is a free tool from AT&T labs, available at `http://www.research.att.com/sw/tools/graphviz/`.

complete, when the expressions are using `//`, `*`, and `[ ]` (predicates) [10]. When disjunctions or DTDs are added then the complexity becomes PSPACE complete, as shown in [11]; and it is even higher when joins are considered too [9]. For XViz we settled on an algorithm for checking the ancestor/descendant and the containment relationships that is quite efficient (it runs in time $O(mn)$ where m and n are the sizes of the two XPath expressions) yet as complete as possible, given the theoretical limitations; the algorithm is adapted from the homomorphism test described in [10].

Related work Hy+ is a query and data visualization tool [8, 7], used for object-oriented data. It is also a graphical editor. For XML languages, several graphical editors have been described. The earliest is for a graphical query language, XML-GL [5]. More recently, a few graphical editors for XQuery have been described. QURSED is a system that includes a graphical editor for XQuery [13], and XQBE is graphical editor designed specifically for XQuery [2].

What sets XViz aside from previous query visualization tools and query editors is its emphasis on finding and illustrating the semantics relationships between XPath expressions. For XML, this has been made possible only recently, through theoretical work that studied the containment problem for XPath expressions, in [9–11].

2 A Simple Example

To illustrate our tool and motivate the work, we show it here in action on a simple example. Consider a file `f.xquery` with the following workload:

```
Q1:

FOR $x in /a/b
WHERE sum($x/c) > 5
RETURN <result> $x/d </result>

Q2:

FOR $u in /a/b[c=6],
    $v in /a/b
WHERE $u/d > $v/c
RETURN $v/d
```

The tool is invoked like this:

```
xviz -i f.xquery -o f.eps -p -q
```

This generates the output file `f.eps`, which is shown in in Fig. 1 (a). Notice that there is one node for each XPath expression in each query, and for

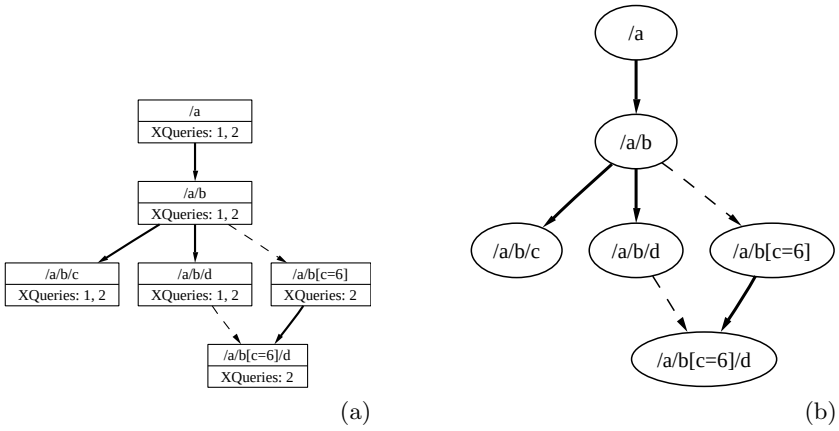

Fig. 1. Example: with XQuery IDs (a) and without (b).

Flag	Meaning	Sample output
`-p`	displays the XPath expression	`/a/b[c=6]/d`
`-q`	displays the query where the expressions occurs	`XQuery: 2, 5, 9`
`-f`	displays for each query the FLWR statement where it occurs	`XQuery: 2(F), 5(W), 9(W)`
`-v`	displays the variable name that is bound to it	`XQuery: 2($x), 5(-), 9($y,-)`
`-b`	brief: do not include prefixes of the XPath expressions	
`-l`	display left-to-right (rather than top-to-bottom)	

Fig. 2. Flags used in conjunction with the `xviz` command.

each prefix of such an expression, with two kinds of edges: solid edges denote ancestor/descendant relationships and dashed edges denote containment relationships.

There are several flags for the command line that control what pieces of information is displayed. The flags are shown in Figure 2. For example, the first one, `-p`, determines only the XPath expression to be displayed, i.e. drops the XQuery identifiers. When used this way on our example, XViz produces the graph in Fig. 1 (b).

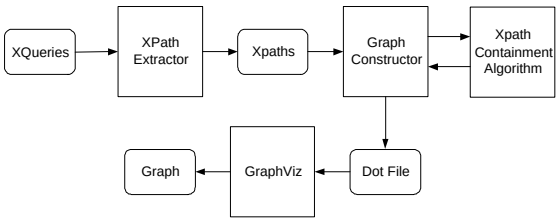

Fig. 3. The System's Architecture

3 Architecture

The overall system architecture is shown in Fig. 3. The input consists of a text file containing a workload of XQuery expressions. The file does not need to contain pure XQuery code, but may contain free text or code in a different programming language, interleaved with XQuery expressions: this is useful for example in cases where the workload is extracted from a document or from an application. The XQuery workload is input to an XPath extractor that identifies and extracts all XPath expressions in the workload. The extractor uses a set of heuristics to identify the XQuery expressions, and, inside them the XPath expressions. Next, the set of XPath expressions are fed into the graph constructor. This makes several calls to the XPath containment algorithm (described in Sec. 4) in order to construct the graph to be displayed. Finally, the graph is displayed using GraphViz. A variety of output formats can be generated by GraphViz: postscript, gif, pdf, etc.

4 Relationships Between XPath Expressions

XViz computes the following two relationships between XPath expressions: ancestor/descendant, and containment. We define them formally below. Notice that both definitions are *semantic*, i.e. independent of the particular syntactic representation of the XPath expression. This is important for XViz applications, since it is precisely these hard to see semantic relationships that are important to show to the user.

We denote with p some XPath expression, and with t some XML tree. Then $p(t)$ denotes the set of nodes obtained by evaluating p on the XML tree t. We shall always assume that the evaluation starts at the root of the XML tree, i.e. all our XPath expressions start with /.

We denote nodes in t with symbols $x, y, \ldots$ If x is a proper ancestor of y then we write $x \ll y$: that is x can be y's parent, or its parent's parent, etc.

Ancestor/Descendant We say that p' and p are in the ancestor/descendant relationship, denoted $p' \ll p$, if for every tree t and for any node $y \in p(t)$ there exists some node $x \in p'(t)$ such that $x \ll y$. Notice that the definition is semantic,

i.e. in order to apply it directly one would need to check all possible XML trees t. We will give below a practical algorithm for checking $\ll$.

Example 1. The following are examples and counterexamples of ancestor/descendant relationships:

$$\begin{aligned}
\texttt{/a/b[c=6]} &\ll \texttt{/a/b[c=6]/d} \\
\texttt{/a/b[c=6]} &\ll \texttt{/a/b[c=6]/d[e=9]/f} \\
\texttt{/a/b} &\not\ll \texttt{/a/b} \\
\texttt{/a//b} &\ll \texttt{/a/b[c=6]/d} \qquad (1) \\
\texttt{/a/b[c=6]} &\not\ll \texttt{/a/b/d} \\
\texttt{/a//b[c//@d=3][@e=5]} &\ll \texttt{a/b[@e=5][@f=7][c/@d=3]/g/h}
\end{aligned}$$

The first two examples should be clear. The third illustrates that the ancestor/descendant relationship is strict (i.e. anti-reflexive: $p \not\ll p$). The next example, (1), shows a particular choice we made in the definition of $\ll$. A node y returned by `/a/b[c=6]/d` always has an ancestor x (namely the `b` node) that is also returned by `/a//b`; but `/a//b` can return nodes that are not ancestors of any node satisfying `/a/b[c=6]/d`. The next two examples further illustrates this point. There are some theoretical arguments in favor of our choice of the definition (the elegant interaction between $\ll$ and $\supseteq$ defined below), but other choices are also possible.

Containment We say that p' contains p, in notation $p' \supseteq p$, if for every XML tree t, the following inclusion holds: $p'(t) \supseteq p(t)$. That is, the set of nodes returned by p' includes all nodes returned by p. Notice that we place the larger expression on the left, writing $p' \supseteq p$ rather than $p \subseteq p'$ as done in previous work on query containment [9–11], because we want in the graph an arrow to go from the larger to the smaller expression.

Example 2. The following illustrate some examples of containment and non-containment:

$$\begin{aligned}
\texttt{/a/b} &\supseteq \texttt{/a/b[c=6]} \\
\texttt{/a//e} &\supseteq \texttt{/a/b[c=6][d=9]/e} \\
\texttt{/a//*/e} &\supseteq \texttt{/a/*//e} \\
\texttt{/a/b[c=6]} &\supseteq \texttt{/a/b[c=6][d=9]} \\
\texttt{/a/b} &\not\supseteq \texttt{/a/b/c}
\end{aligned}$$

Here too, the definition is semantic: we will show below how to check this efficiently. Notice that it is easy to check equivalence between XPath expressions by using containment: $p \equiv p'$ iff $p \supseteq p'$ and $p' \supseteq p$.

4.1 Reducing Ancestor/Descendant to Containment

The two relationships can be reduced to each other as follows:

$$p' \ll p \Longleftrightarrow p'//* \supseteq p$$
$$p' \supseteq p \Longleftrightarrow p'/a \ll p/a/*$$

Here a is any tag name that does not occur in p'. We use the first reduction in order to compute $\ll$ using an algorithm for $\supseteq$. We use second reduction only for theoretical purposes, to argue that all hardness results for $\supseteq$ also apply to $\ll$. For example, for the fragment of XPath described in [10], checking the relationship $\ll$ is co-NP complete.

4.2 Computing the Graph

XViz uses the relationships $\ll$ and $\supseteq$ to compute and display the graph. A relationship $p' \ll p$ will be displayed with a solid edge, while $p' \supseteq p$ is displayed with a dashed edge.

Two steps are needed in order to compute the graph. First, identify equivalent expressions and collapse them into a single graph node. Two XPath expressions are equivalent, $p \equiv p'$ if both $p \supseteq p'$ and $p' \supseteq p$ hold. Once equivalent expressions are identified and removed, only $\supset$ relationships remain between XPath expressions.

Second, decide which edges to represent. In order to reduce clutter, redundant edges need not be represented. An edge is *redundant* if it can be inferred from other edges using one of the four implications below:

$$p_1 \supset p_2 \wedge p_2 \supset p_3 \Longrightarrow p_1 \supset p_3$$
$$p_1 \ll p_2 \wedge p_2 \ll p_3 \Longrightarrow p_1 \ll p_3$$
$$p_1 \ll p_2 \wedge p_2 \supset p_3 \Longrightarrow p_1 \ll p_3$$
$$p_1 \supset p_2 \wedge p_2 \ll p_3 \Longrightarrow p_1 \ll p_3$$

The first two implications state that both $\ll$ and $\supset$ are transitive. The last two capture the interactions between them.

Redundant edges can be naively identified with three nested loops, iterating over all triples (p_1, p_2, p_3) and marking the edge on the right hand side as redundant whenever the conditions on the left is satisfied. This method takes $O(n^3)$ steps, where n is the number of XPath expressions. We will discuss a more efficient way in Sec. 6.

5 An Application

We have experimented with XViz applied to three different workloads: the XMark benchmark [12], the XQuery Use Cases [6], and the XMach benchmark [4]. We describe here XMark only, which is shown in Fig. 4. The other two are similar: we show a fragment of the XQuery Use cases in Fig. 5, but omit XMach for lack of space.

The result of applying XViz to the entire XMark benchmark[3] is shown in Fig. 4. It is too big to be readable in the printed version of this paper, but can be magnified when read online.

Most of the relationships are ancestor/descendant relationships. The root node `/` has one child, `/site`, which in turn has the following five children:

```
/site/people
/site//item
/site/regions
/site/open_auctions
/site/closed_auctions
```

Four of them correspond to the four children of `site` in the XML schema, but `/site//item` does not have a correspondence in the schema. We emphasize that, while the graph is somewhat related to the XML schema, it is different from the schema, and precisely these differences are interesting to see and analyze.

For example, consider the following chain in the graph:

```
/site ≪ /site//item
      ⊃ /site/regions//item
      ⊃ /site/regions/europe/item
      ≪ /site/regions/europe/item/name
```

Or consider the following two chains at the top of the figure, that start and end at the same node (showing that the graph is a DAG, not a tree):

```
/site/people/person ⊃ /site/people/person[@id='person0']
                    ≪ /site/people/person[@id='person0']/name
/site/people/person ≪ /site/people/person/name
                    ⊃ /site/people/person[@id='person0']/name
```

They both indicate relationships between XPath expressions that can be of great interest to an administrator, depending on her particular needs.

For a more concrete application, consider the expressions:

```
/site/people/person/name
/site/people/person[@id='person0']/name
```

The first occurs in XQueries 8, 9, 10, 11, 12, 17 is connected by a dotted edge (i.e. $\supset$) to the second one, which also occurs in XQuery 1. Since they occur in relatively many queries, are good candidates for building an index. Another such candidate consists of `p` = `/site/closed_auctions/closed_auction`, which occurs in queries 5, 8, 9, 15, 16, together with several descendants, like `p/seller`, `p/price`, `p/buyer`, `p/itemref`, `p/annotation`.

[3] We omitted query 7 since it clutters the picture too much.

6 Implementation

We describe here the implementation of XViz, referring to the Architecture in Fig. 3.

6.1 The XPath Extractor

The XPath extractor identifies XQuery expressions in a text and extracts as many XPath expressions from these queries as possible. It starts by searching for the keywords `FOR` or `LET`. The following text is then examined to see if a valid XQuery expression follows. We currently parse only a fragment of XQuery, without nested queries or functions. The grammar that we support is described in Fig. 6.

In this grammar, each Variable is assumed to start with a $ symbol and each XPathExpr is assumed to be a valid XPath expression. XPathText is a body of text, usually a combination of XML and expressions using XPaths, that we can extract any number of XPath expressions from. After an entire XQuery has been parsed, each XPath Expression is expanded by replacing all variables with their declared expressions. Once all XPath expressions have been extracted from a query, the Extractor continues to step through the text stream in search of XQuery expressions.

6.2 The XPath Containment Algorithm

The core of XViz is the XPath containment algorithm, checking whether $p' \supseteq p$ (recall that this is also used to check $p' \ll p$, see Sec. 4.1). If the XQuery workload has n XPath expressions, then the containment algorithm may be called up to $O(n^2)$ times (some optimizations may reduce this number however, see below), hence we put a lot of effort in optimizing the containment test. Namely, we checked containment using homomorphisms, by adapting the techniques in [10]. For presentation purposes we will restrict our discussion to the the XPath fragment consisting of tags, wildcards $*$, /, //, and predicates [ ], and mention below how we extended the basic techniques to other constructs.

Each XPath expression p is represented as a tree. A node, x, carries a label $\text{LABEL}(x)$, which can be either a tag or $*$; $\text{NODES}(p)$ denotes the set of nodes. Edges are of two kinds, corresponding to / and to // respectively, and we denote $\text{EDGES} = \text{EDGES}_{/} \cup \text{EDGES}_{//}$.

A *homomorphism* from p' to p is a function from $\text{NODES}(p')$ to $\text{NODES}(p)$ that maps each node in p' to a matching node in p (i.e. it either has the same label, or the node in p' is $*$), maps an /-edge to an /-edge, and maps a //-edge to a path, and maps the return node in p' to the return node in p. Fig. 7 illustrates a homomorphism from $p' = /a/a[.//b]/*[c]//a/b$ to $p = /a/a/[.//c]/d[c]//a[a]/b$. Notice that the edge $a//b$ is mapped to the path $a/d//a/b$.

If there exists a homomorphism from p' to p then $p' \supseteq p$. This allows us to check containment by checking whether there exists homomorphism. This is done bottom-up, using dynamic programming. Construct a boolean table $\mathcal{C}$

where each entry $\mathcal{C}(x, y)$ for $x \in \text{NODES}(p), y \in \text{NODES}(p')$ contains 'true' iff there exists a homomorphism mapping y to x. The table $\mathcal{C}$ can be computed bottom up since $\mathcal{C}(x, y)$ depends only on the entries $\mathcal{C}(x', y')$ for y' a child of y and x' a child or a descendant of x. More precisely, $\mathcal{C}(x, y)$ is true iff $\text{LABEL}(y) = *$ or $\text{LABEL}(y) = \text{LABEL}(x)$ and, for every child y' of y the following conditions holds. If $(y, y') \in \text{EDGES}_/(p')$ then $\mathcal{C}(x', y')$ is true for some /-child of x:

$$\bigvee_{(x,x') \in \text{EDGES}_/(p)} \mathcal{C}(x', y')$$

If $(y, y') \in \text{EDGES}_/(p')$ then $\mathcal{C}(x', y')$ is true for some descendant x' of x:

$$\bigvee_{(x,x') \in \text{EDGES}^+(p)} \mathcal{C}(x', y') \tag{2}$$

Here $\text{EDGES}^+(p)$ denotes the transitive closure of $\text{EDGES}(p)$. This can be directly translated into an algorithm of running time $O(|p|^2|p'|)$.

Optimizations We considered the following two optimizations.

The first addresses the fact that there are some simple cases of containment that have no homomorphism. For example there is no homomorphism from $/a//*/b$ to $/a/*//b$ (see Figure 8 (a)) although the two expressions are equivalent. For that we remove in p' any sequence of $*$ nodes connected by / or // edges and replace them with a single edge, carrying an additional integer label that represents the number of $*$ nodes removed. This is shown in Figure 8 (b). The label thus associated to an edge (y, y') is denoted $k(y, y')$. For example $k(y, y') = 1$ in Fig. 8 (b).

The second optimization reduces the running time to $O(|p||p'|)$. For that, we compute a second table, $\mathcal{D}(x, y')$, which records whenever there exists a descendant x' of x s.t. $\mathcal{C}(x', y')$ is true. Moreover, $\mathcal{D}(x, y')$ contains the actual distance from x to x'. Then, we can avoid a search for all descendants x' and replace Eq.(2) with the test $'D(x, y') \geq 1 + k(y, y')$. Both $\mathcal{C}(x, y)$ and $\mathcal{D}(x, y)$ can now be computed bottom up, in time $O(|p||p'|)$, as shown in Algorithm 1.

Other XPath Constructs Other constructs, like predicates on atomic values, `first()`, `last()` etc, are handled by XViz by extending the notion of homomorphism in a straightforward way. For example a node labeled `last()` has to be mapped into a node that is also labeled `last()`. Additional axes can be handled similarly. The existence of a homomorphism continues to be a sufficient, but not necessary condition for containment.

6.3 The Graph Constructor

The Graph Constructor takes a set of n XPath expressions, $p_1, \ldots, p_n$, computes all relationships $\ll$ and $\supseteq$, eliminates equivalent expressions, then computes a minimal set of solid edges (corresponding to $\ll$) and dashed edges (corresponding to $\supseteq$) needed to represent all $\ll$ and $\supseteq$ relationships, by using the four implications in Sec. 4.2.

Algorithm 1 Find homomorphism $p' \rightarrow p$

1: **for** x in NODES(p) **do** {The iteration proceeds bottom up on nodes of p}
2: **for** y in NODES(p') **do** {The iteration proceeds bottom up on nodes of p'}
3: **compute** $\mathcal{C}(x,y) = (\text{LABEL}(y) = \text{“}*\text{”} \vee \text{LABEL}(x) = \text{LABEL}(y)) \wedge$
4: $\bigwedge_{(y,y') \in \text{EDGES}_{/}(p')} (\bigvee_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{C}(x',y')) \wedge$
5: $\bigwedge_{(y,y') \in \text{EDGES}_{//}(p')} (\mathcal{D}(x,y') \geq 1 + k(y,y'))$
6: **if** $\mathcal{C}(x,y)$ **then**
7: $d = 0$;
8: **else**
9: $d = -\infty$
10: **compute** $\mathcal{D}(x,y) = \max(d, 1 + \max_{(x,x') \in \text{EDGES}_{/}(p)} \mathcal{D}(x',y),$
11: $1 + \max_{(x,x') \in \text{EDGES}_{//}(p)} (k(x,x') + \mathcal{D}(x',y)))$
12: **return** $\mathcal{C}(\text{ROOT}(p), \text{ROOT}(p'))$

A naive approach would be to call the containment test $O(n^2)$ times, in order to compute all relationships[4] $p_i \ll p_j$ and $p_i \supseteq p_j$, then to perform three nested loops to remove redundant relationships (as explained in Sec. 4.2), for an extra $O(n^3)$ running time.

To optimize this, we compute the graph G incrementally, by inserting the XPath expressions $p_1, \ldots, p_n$, one at a time. At each step the graph G is a DAG, whose edges are either of the form $p_i \ll p_j$ or $p_i \supset p_j$. Suppose that we have computed the graph G for $p_1, \ldots, p_{k-1}$, and now we want to add p_k. We search for the right place to insert p_k in G, starting at G's roots. Let G_0 be the roots of G, i.e. the XPath expressions that have no incoming edges. First determine if p_k is equivalent to any of these roots: if so, then merge p_k with that root, and stop. Otherwise determine whether there exists any edge(s) from p_k to some XPath expression(s) in G_0. If so, add all these edges to G and stop: p_k will be a new root in G. Otherwise, remove the root nodes G_0 from G, and proceed recursively, i.e. compare p_k with the new of roots in $G - G_0$, etc. When we stop, by finding edges from p_k to some p_i, then we also need to look one step “backwards” and look for edges from any parent of p_i to p_k. While the worst case running time remains $O(n^3)$, with $O(n^2)$ calls to the containment test, in practice this performs much better.

7 Conclusions

We have described a tool, XViz, to visualize sets of XPath expressions, together with their relationships. The intended use for XViz is by an XML database administrator, in order to assist her in performing various tasks, such as index selection, debugging, version management, etc. We put a lot of effort in making the tool scalable (process large numbers of XPath expressions) and usable (accept flexible input).

[4] Recall that $p_i \ll p_j$ is tested by checking the containment $p_i//* \supseteq p_j$.

We believe that a powerful visualization tool has great potential for the management of large query workloads. Our initial experience with standard workloads, like the XMark Benchmark, gave us a lot of insight about the structure of the queries. This kind of insight will be even more valuable when applied to workloads that are less well designed than the publicly available benchmarks.

Acknowledgments This research was partially supported by NSF Grant IIS-0140493, a gift from Microsoft, and by Suciu's NSF CAREER Grant 0092955 and Alfred P. Sloan Research Fellowship.

References

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 496–505. Morgan Kaufmann, 2000.
2. E. Augurusa, D. Braga, A. Campi, and S. Ceri. Design of a graphical interface to XQuery. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 226–231, 2003.
3. P. Bohannon, J. Freire, P. Roy, and J. Simeon. From xml schema to relations: A cost-based approach to xml storage. In *ICDE*, 2002.
4. T. Böhme and E. Rahm. Multi-user evaluation of XML data management systems with XMach-1. In *Proceedings of the Workshop on Efficiency and Effectiveness of XML Tools and Techniques (EEXTT)*, pages 148–158. Springer Verlag, 2002.
5. S. Ceri, S. Comai, E. Damiani, P. Fraternali, and S. Paraboschi. XML-gl: a graphical language for querying and restructuring XML documents. In *Proceedings of WWW8*, Toronto, Canada, May 1999.
6. D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: an XML query language, 2001. available from the W3C, `http://www.w3.org/TR/query`.
7. M. Consens, F. Eigler, M. Hasan, A. Mendelzon, E. Noik, A. Ryman, and D. Vista. Architecture and applications of the hy+ visualization system. *IBM Systems Journal*, 33:3:458–476, 1994.
8. M. P. Consens and A. O. Mendelzon. Hy: A hygraph-based query and visualization system. In *Proceedings of 1993 ACM SIGMOD International Conference on Management of Data*, pages 511–516, Washington, D. C., May 1993.
9. A. Deutsch and V. Tannen. Optimization properties for classes of conjunctive regular path queries. In *Proceedings of the International Workshop on Database Programming Lanugages*, Italy, Septmeber 2001.
10. G. Miklau and D. Suciu. Containment and equivalence of an xpath fragment. In *Proceedings of the ACM SIGMOD/SIGART Symposium on Principles of Database Systems*, pages 65–76, June 2002.
11. F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *International Conference on Database Theory*, 2003.
12. A. Schmidt, F. Waas, M. Kersten, D. Florescu, M. Carey, I. Manolescu, and R. Busse. Why and how to benchmark XML databases. *Sigmod Record*, 30(5), 2001.
13. V. V. Yannis Papakonstantinou, Michalis Petropoulos. QURSED: querying and reporting semistructured data. In *Proceedings ACM SIGMOD International Conference on Management of Data*, pages 192–203. ACM Press, 2002.

Fig. 4. XViz showing the entire XMark Benchmark workload.

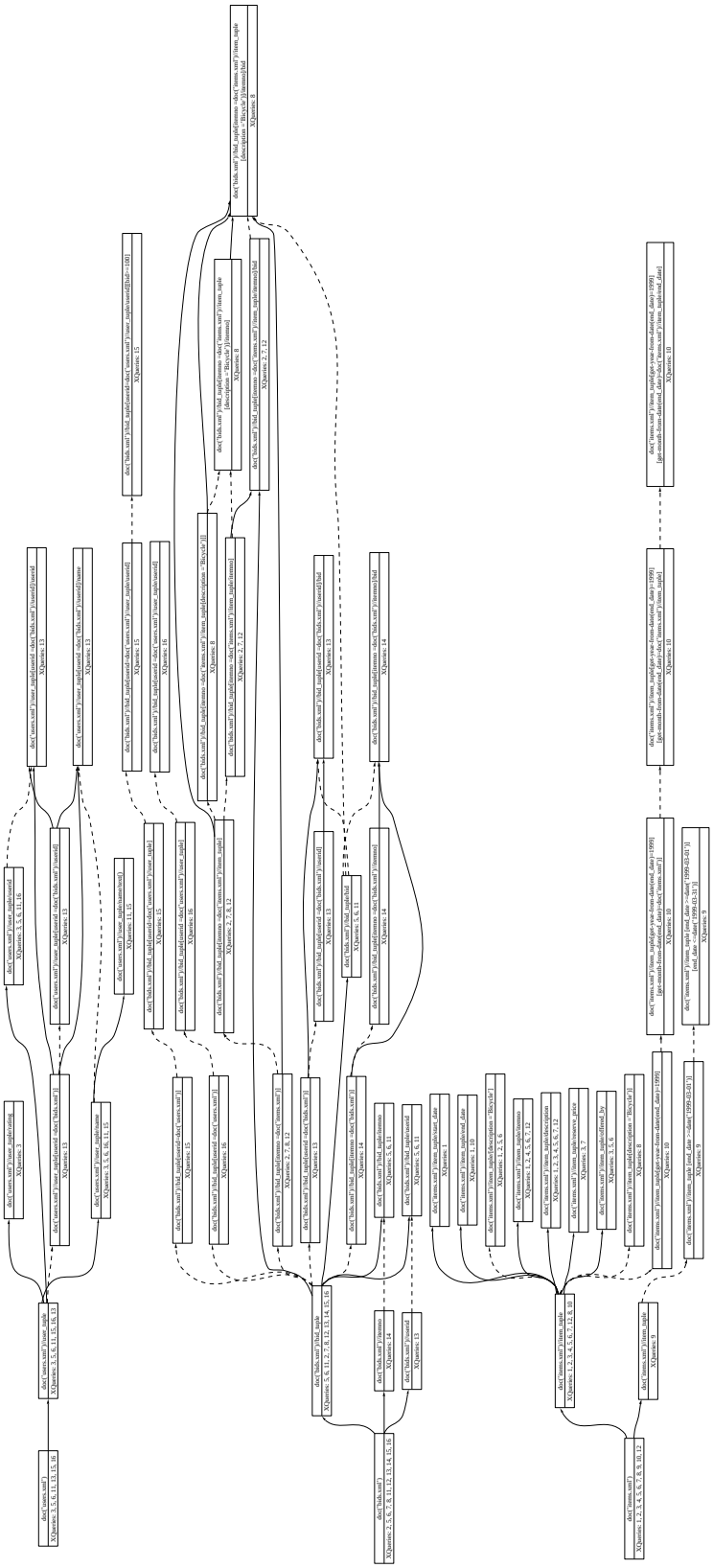

Fig. 5. XPath Expressions from the "R" Section of the XQuery Use Cases.

FlwrExpr ::= (ForClause | letClause)+ whereClause? returnClause
ForClause ::= 'FOR' Variable 'IN' Expr (',' Variable IN Expr)*
LetClause ::= 'LET' Variable ':=' Expr (',' Variable := Expr)*
WhereClause ::= 'WHERE' XPathText
ReturnClause ::= 'RETURN' XPathText
Expr ::= XPathExpr | FlwrExpr

Fig. 6. Simplified XQuery Grammar

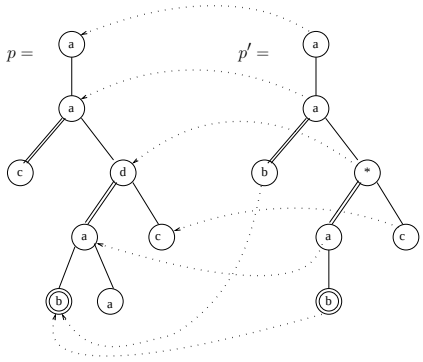

Fig. 7. Two tree patterns p, p' and a homomorphism from p' to p, proving $p' \supseteq p$.

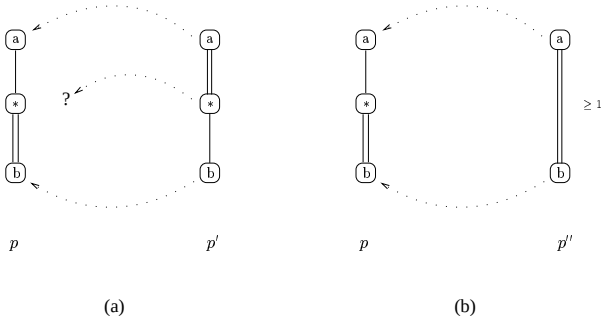

Fig. 8. (a) Two equivalent queries p, p' with no homomorphism from p' to p; (b) same queries represented differently, and a homomorphism between them.