

Improving XPath Evaluation with Strategies

Karl J. Lieberherr Jeffrey Palm

*Northeastern University
Boston, MA*

Abstract

XPath is the defacto navigation language for XML documents conforming to DTD and it is important to have efficient evaluation and checking techniques available for XPath. Traversal strategies, a component of Adaptive Programming, are a well studied navigation language for objects conforming to class graphs and an efficient evaluation technique has been developed in the previous millenium. Traversal strategies don't deal with the full generality of XPath but instead they focus on ancestor-descendent (ad) edges. Indeed, a traversal strategy is a graph consisting only of ad edges with optional negative constraints on the edges.

While it has been widely acknowledged in the data base community that DTD may speed up XPath evaluation, no paper has shown an exponential improvement by using the DTDs. We show an infinite sequence of XPath expression/DTD/document triples so that evaluation using the meta information in the DTDs is exponentially faster. We draw a strong connection between traversal strategy evaluation and XPath evaluation and show how this leads to comprehensive logical optimization techniques for XPath. Our evaluation is optimal in the sense that by visiting one node less than our approach visits would lead to wrong results. Finally, we present empirical results of our speed up and explore various classes of XML documents that can benefit from this type of evaluation.

We believe that the automata-theoretic algorithmic ideas presented in this paper should be an important building block of any efficient XPath evaluator.

Key words: AspectJ, Demeter, pointcut designators, traversal strategies
PACS:

1 Introduction

XPath is a language for navigating an XML tree and returning a set of answer nodes. These expressions are used in XQuery [15], XML Schema [1], XLink [18], and XPointer [17]. In addition they are often used to query large XML databases such as the PIR-International Protein Sequence Database [2].

XPath	Adaptive Programming
DTD	Class graph
XML document	Object graph
XPath expression	Strategy

Table 1
Relating XPath to Adaptive Programming

Document Type Definitions (DTDs) provide a means of constraining XML documents by defining their structure. A DTD is commonly modeled as a number of expressions $n \rightarrow R$ where R is a regular expression describing the children that are allowed to appear directly beneath nodes labeled n .

Databases are more commonly being stored as XML, and database queries are then being carried out through XPath evaluation. It is therefore imperative for these queries to perform efficiently. That said, a common problem when evaluating XPath queries on XML documents is that, at a given node, we don't necessarily know what are the "meaningful" nodes to explore next – i.e. what nodes could possibly lead to the result set of query. Luckily, there is an analogous problem in traversing object graphs in Adaptive Programming (AP).

AP is a technique for Object-Oriented programming in which traversals of objects are specified in *strategies* [25]. In this setting a *class graph* specifies the layout of objects that are arranged in an *object graph*. A user may then specify traversals over this object graph by writing *strategies*. These strategies are written as a graph where each edge defines a set of paths that must be visited in a traversal. If we view the class graph and strategy as automata we can create a *traversal graph* from the cross product of the strategy and class graph. Thus, we can ignore paths that could never lead to an accepting state, and the object graph can be efficiently traversed [23].

In this paper we focus on improving the performance of XPath queries for XML documents whose DTD is available by making use of this meta information. We draw the following connection between the XML and AP worlds, also shown in Table 1: XML documents can be modeled by unordered trees; object graphs are unordered graphs. DTDs define the structure of XML documents; class graphs define the structure of object graphs. XPath expressions define the nodes to visit when navigating an XML tree; strategies define the nodes to visit when navigating an object graph.

We first show that evaluating XPath queries without meta-information may be exponentially slower than with meta-information. We then show how to use meta-information to speed up query evaluation.

2 Definitions

We adopt Wood’s model of DTDs [31]. For a DTD D this model consists of a finite alphabet, Σ , a root type, denoted $\text{root}(D)$, and a mapping that associates with each $a \in \Sigma$ a regular expression of Σ . This is most easily written as a EBNF grammar where the production right-hand sides are regular expressions and the alphabet elements are non-terminals.

We use the model used by Miklau *et al.* for XML documents as trees over infinite alphabets [28]. We also use the fragment of XPath known as $\text{XP}^{\{*,//\}}$, which consists of node tests, the child axis ($/$), the descendant axis ($//$), and wildcards ($*$). This model does not capture the full XPath, but is sufficient for analyzing the speed we can gain over structural queries. A full semantics of an XPath query is given in [30].

The following is adapted from [24]. A *class graph* consists of a set C of classes; a set E of field names; and for each $e \in E$ a relation denoted e (“has part named e ”) on classes, and a reflexive transitive relation \leq on classes (“is a subclass of”).

If C is a class graph, then an *object graph* for C consists of:

- (1) a set O (“of objects”),
- (2) a map $\text{class}: O \rightarrow C$, and
- (3) for each $e \in E$, a relation denoted e on O such that if $e(o_1, o_2)$, then

$$\text{class}(o_1) (\leq e \geq) \text{class}(o_2)$$

We say that o is of type c when $\text{class}(o) \leq c$.

The traversal of an edge labeled e corresponds to retrieving the value of the e field. Condition 3 captures the notion that every edge in the object graph is an image of a has-as-part edge in the class graph: There is an edge $e(o_1, o_2)$ in O only when there exist classes c_1 and c_2 such that o_1 is of type c_1 , c_1 has an e -part of type c_2 , and o_2 is of type c_2 , that is,

$$\text{class}(o_1) (\leq c_1 e c_2 \geq) \text{class}(o_2)$$

The valid traversals of a class graph is given in terms of FIRST sets. For each pair of classes c and c' we have an edge $e \in \text{FIRST}(c, c')$ iff it is possible for an object of class c to reach an object of type c' by a path beginning with an edge e . Precisely, $\text{FIRST}(c, c') = \{ e \in E \mid \text{there exists an object graph } O \text{ of } C \text{ and object } o \text{ and } o' \text{ such that:}$

- (1) $\text{class}(c) = c$,

- (2) $\text{class}(o') \leq c'$,
- (3) $o \in O * e'$ }

The last condition says that there is a path from o to o' in the object graph, consisting of an edge labeled e , followed by any sequence of edges in the graph. Furthermore, Wand and Lieberherr show that

$$\text{FIRST}(c, c') = \{e | c (\leq e \geq) (\leq C \geq)^* \leq c'\}.$$

That is, an object of class c' is reachable from an object of class c starting with edge e if we can follow e so some object o' ; then we can follow hasa-edges to an object of type c' . So, a traversal of an object graph is then specified transitively as FIRSTsets.

Traversals may be implemented efficiently using strategy graphs. These graphs lay out a road map for traversing only those edges of an object graph that can lead to successful endpoints. Formally, a strategy graph is given by a set of states Q , a transition relation S on states, a map $\text{class} : Q \rightarrow C$, a set $QI \subset Q$ of initial states, and a set $QF \subset Q$ of final states. A complete discussion of strategy graphs is found in [26], but the important point of these graphs is that we can use them to efficiently traverse object graphs.

A *path* in a graph is a sequence $v_1 \cdots v_n$ where v_1, \dots, v_n are nodes of the graph; and $v_i \rightarrow v_{i+1}$ is an edge of the graphs for all $i \in 1 \dots n - 1$. We call v_1 and v_n the *source* and *target* nodes of the path, respectively.

A strategy, then, selects valid paths in an object graph, and uses information from that object's class graph to prune out paths that could never lead to a successful result.

To execute an XPath query efficiently we can compute the corresponding strategy graph and let this graph prune away paths that could never lead us to a successful outcome. We show this translation in Section 4, but first we give a motivating example showing that query evaluation without meta information may visit exponentially many nodes.

We use the term *meta information* to refer to any information that describes the structure of other information. The meta information of an XML document is its DTD; the meta information of an object graph is its class graph.

3 Example

The goal of this paper is to show that strategies can be used to speed up query evaluation. As Gottlob *et al.* point out, the way XPath is defined motivates

an implementation approach that leads to highly inefficient (exponential-time) XPath processing [20]. So, in this section we show that for some queries and documents, an evaluation without meta information may visit exponentially many nodes. To do so, we first define a *naive evaluation* of a query to be a complete traversal of a document. The key here is that we choose an arbitrary but consistent order to visit the children of a given node. So, consider the DTD D_n

$$\begin{aligned}
\text{root} &\rightarrow a_0 \\
a_0 &\rightarrow a_1 b_0 \\
b_0 &\rightarrow a_1 \\
a_1 &\rightarrow a_2 b_1 \\
b_1 &\rightarrow a_2 \\
&\vdots \\
b_{n-1} &\rightarrow a_n \\
a_n &\rightarrow \#\text{PCDATA}
\end{aligned}$$

and a query

$$Q_n = /root//a_0//b_0//\dots//a_n. \quad (1)$$

A document satisfying this DTD is denoted P_n , and one such document for $n = 3$ has the structure shown in Figure 1. Intuitively a “smart” traversal of this document simply proceeds down the left-hand branch of the tree, and this is shown as the ‘double-circles’ in Figure 1. But, if we choose a right-to-left ordering to visit the children of a node we will visit all nodes before arriving at our target node, a_n . Without loss of generality, we could simply swap the a and b in every $a_i \rightarrow a_{i+1} b_i$ production to $a_i \rightarrow b_i a_{i+1}$ if we had chosen a left-to-right ordering. A naive navigation would, then, visit all the nodes of this tree.

We state this lower bound formally through Theorem 1.

Theorem 1 *There is a sequence of query/DTD/document triples (Q_n, D_n, P_n) such that P_n conforms to D_n , $|Q_n| = O(n)$, $|D_n| = O(n)$, and $|P_n| = o(2^n)$, and so that the naive evaluation will visit $o(2^n)$ nodes in P_n while the meta-information-based evaluation will visit $O(n)$ nodes in P_n .*

Proof 1 *Consider the DTD and query above. For an $n > 0$, the DTD has $5n+4$ nodes, and any satisfying document will have $2^{n+1}+3$ nodes. Additionally if we choose a left-to-right traversal, the naive evaluation will visit all $2^{n+1}+3$ nodes before arriving at the one node satisfying the query, a_n .*

Proof 2 *The root must have one child, a_0 , since $n > 0$ so we visit the root. We then proceed by induction on the index of the current node, i . If $i = n$ then we visit 2 nodes – a_i and its one child b_i . For $i \neq n$, a_i has two children, a_{i+1} and b_{i+1} . The strategy graph does not permit a move to a_{i+1} , so our only move is to visit 2 more nodes – b_{i+1} and its only child a_{i+1} ; both allowed by the strategy.*

So, we’ve shown the lower bound of naive evaluation can visit exponentially many nodes, where as evaluation with a strategy may visit just a linear number of nodes.

4 Translation

To use meta-information to speed up query evaluation we must translate our DTD into a class graph and our query into a strategy graph. We show how to do both in this section.

4.1 DTD translation

We begin by defining a *flat* class graph.

Definition 1 *A class graph is flat iff all nodes with incoming isa-edges have no outgoing hasa-edges.*

Our translation from DTD to class graph is in two steps. In the first step we convert a DTD into a valid *flat* class graph. For the second step we note that there is no notion of inheritance in DTDs, but a closely related concept of alternation. So, for the second step we remove all isa-edges from this flat graph. We show that hasa-edges are enough to traverse this graph efficiently. Additionally, we prove that the traversal guided by isa-edge and hasa-edge information is equivalent to the traversal guided by just hasa-edges after removing the isa-edges.

A class graph can be specified using the LL(1) language called a *class dictionary* consisting of isa-edge productions

$$c_0 : c_1 \mid \cdots \mid c_n$$

that state classes c_1 through c_n are subclasses of c_0 ; hasa-edge productions

$$c_0 = c_1 \cdots c_m$$

that states class c_0 contains objects of types c_1 through c_n ; and repetition productions

$$c_0 \sim \cdots \{ c_1 \} \cdots$$

that states class c_0 contains zero or more objects of type c_1 .

A DTD is modelled as productions of the form

$$n \rightarrow R$$

where n is a node name and R is a regular expression of node names containing sequence ($n_1 \cdots n_m$), alternation ($|$) and repetition ($*$). Hence, we can repeatedly rename nested terms in the DTD using a translation, T , until every production is either a sequence, alternation, or repetition. At this point, this grammar is a valid class dictionary. Explicitly, our translation of these terms, is

$$\begin{aligned} T(n_0 \rightarrow n_1 | \cdots | n_n) &\Downarrow \begin{aligned} n_0 &\rightarrow n'_0 \\ n'_0 &\rightarrow n_1 | \cdots | n_n \end{aligned} \\ T(n_0 \rightarrow \cdots n_1^* \cdots) &\Downarrow \begin{aligned} n_0 &\rightarrow \cdots n'_0 \cdots \\ n'_0 &\rightarrow n_1^* \end{aligned} \end{aligned}$$

where n_i are node names.

DTD node names must appear exactly once on the left side of productions, and by the definition of class graphs (the class names are a set), class names must appear exactly once on the left side of class dictionary productions. So, clearly a class graph obtained from translating a DTD D , $G = T(D)$, is flat.

Removing all the alternation (i.e. $|$'s) in the DTD has the effect of pushing all the hasa-edges into concrete classes. We must now show that for a given strategy graph the traversal graph produced from a class graph without inheritance edges, G , is equivalent to that produced from the class graph with inheritance, G' . So, given a class graph $G = (C, E)$, our translated class graph is $G' = (C, E')$ where

$$E' = \left\{ e \mid \begin{aligned} &e \in E, \text{ or} \\ &e = (A, B) \text{ and } (B, A) \in C \text{ is an isa-edge} \end{aligned} \right\}.$$

Intuitively this says that all hasa-edges in G are in G' , and we reverse all hasa-edges in G and turn them into isa-edges in G' . Additionally, we reason in terms of FIRST sets and show that $\text{FIRST}_G(c, c')$ is equal to the FIRST set of the class graph without inheritance – $\text{FIRST}_{G'}(c, c')$. We formally state this in Theorem 2.

Theorem 2 For each pair of classes c and c' in a flat class graph G , if $G' = T(G)$ then $\text{FIRST}_G(c, c') = \text{FIRST}_{G'}(c, c')$.

Proof 3 If $e \in \text{FIRST}_G(c, c')$ then $c (\leq e \geq) (\leq C \geq)^* \leq c'$. So, there is a hasa-edge from c to some node d .

- In the case that $d = c$ the theorem is trivially-true.
- Otherwise there is some node g where $d(\leq C \geq)^*g$ and $c' \leq g$, therefore $c'(\leq C \geq)^*g$. In our translation every C -edge (A, B) in G is reversed in G' as an hasa-edge. So, there is a path $d (\leq e \geq) c'$.

4.2 XPath translation

We will translate a query into a strategy graph, and this strategy graph is specified using DJ [27]. DJ is an implementation of strategies and provides a concise syntax for expressing strategies. So, for every statement $A//B$ in the query we add an edge from A to B in the strategy graph specified as

$$A \rightarrow B.$$

For every statement A/B in the query we add an edge from A to B in the strategy graph and a constraint that mandates traversals pass through a hasa-edge from A to B . A full treatment of constraints in strategy graphs is found in [26], but we can simply specify it using DJ as

$$A \text{ only-through } * \rightarrow * B$$

which requires that a satisfying traversal pass through any hasa-edge of A to B .

The semantics of XPath queries are given in terms of sets of nodes [30]; while the semantics of strategies are given in terms of sets of paths [26]. So, we augment the meaning of a strategy so that if a strategy specifies paths p_1, \dots, p_n , then we are only interested in the target nodes. Thus, when we specify a query as a strategy, the result of that query is then the set of target nodes in a successful traversal. To see that our transformation is valid, we just note that the target nodes of the paths selected by that strategy are exactly those nodes selected by the query. This comes straight from the semantics of each.

From the definition of the descendent axis, the query $A//B$ will evaluate to a set containing all nodes B reachable from A ; starting at an object of type A the strategy $A \rightarrow B$ will visit all nodes of type B that are reachable from objects of type A . For the child axis, A/B evaluates to a set containing all nodes B connected by an outgoing edge from A ; starting at an object of type A , the

strategy `A only-through *->* B` will visit all objects of type `B` connected by a `has-a`-edge (i.e. contained) in objects of type `A`.

5 Main Results

Our main result is the realization that meta information can theoretically speed up XPath queries. But, we also want to show that (1) we can speed up evaluation implementations in practice, and (2) there are many relevant DTDs for which this type of speed up applies. To show (1) we will present empirical results gathered from evaluating the example in Section 3; for (2) we will examine some relevant DTDs and XPath queries found in practice and try to characterize general classes of DTD/query pairs that can benefit from strategies.

5.1 Empirical Results

We compared the running time of three different implementations of the `org.w3c.dom.traversal.NodeIterator` interface [12]. An implementation of this interface simply steps through a set of nodes and is often used as the result of evaluating an XPath query, and we took into account the time to create an instance as well as traverse the document. Each of the following implementations correctly implemented XPath semantics but chose a different strategy for exploring a document:

- **StrategyNodeIterator**: Traversed those nodes selected from a strategy created from the given query.
- **NaiveNodeIterator**: Traversed the document naively, choosing the next node to explore in a left-to-right scan of the current node.
- **DTMNodeIterator**: The Apache implementation of `NodeIterator` found in the current Java Development Kit from Sun [8].

Some points are worth noting first, though. The first two only implement the XPath fragment of $XP^{\{*,//\}}$, whereas the third is an implementation of the full XPath. That said, we cannot fairly compare the running times of the third to the first two, because there is a considerable amount of book-keeping done here that is not done in the first two in order to implement the full specification. However, the exponential blowup in the time is worth noting, so we include those results.

The `NaiveNodeIterator` and `StrategyNodeIterator` are identical implementations except that they differ in what node to visit next. `NaiveNodeIterator`

n	Strategy	Naive	DTM
5	3.3	3.1	11.1
6	4.4	11.8	19.6
7	2.7	1.7	29.6
8	2.1	3.8	27.0
9	2.6	3.9	16.2
10	8.70	6.0	29.2
11	7.31	39.7	65.8
12	2.52	20.0	115.5
13	5.13	40.9	356.3
14	2.54	82.2	669.9
15	2.65	165.2	1480.3
16	3.26	806.1	3853.1
17	2.87	1715.2	22104.8

Table 2

Time (ms) to execute Q_n on a D_n -conforming document.

always choses the first valid child according to a query in a left-to-right scan of the current nodes children; where `StrategyNodeIterator` will choose the next node suggested by a strategy created from a query. This strategy is computed using the APLib [3] by translating the DTD into a class graph, the query into a traversal specification. This translation is done using the Demeter software suite [6] ¹.

Table 1 shows these results shows the time in milliseconds to execute the query Q_n from the previous section on a document conforming to the DTD D_n for the three implementations. Additionally 1 shows the number of nodes visited by the strategy and naive implementations.

Again, we conclude two things from these results. First, comparing the naive implementation and the strategy implementation allows us to compare the practical cost of structurally traversing XML documents. The implementation using strategies stayed relatively constant over a varying n whereas the naive implementation increased exponentially. This indicates that, in practice, the cost of structurally traversing documents is significant and can be alleviated by using strategies. Secondly, we cannot compare the running time of the `DTMNodeIterator` with the other two because it fully-implements XPath

¹ This translation implementation is available at <http://www.ccs.neu.edu/home/jpalm/dtd>

n	Strategy	Naive
5	11	380
6	13	764
7	15	1532
8	17	3068
9	19	6140
10	21	12284
11	23	24572
12	25	49148
13	27	98300
14	29	196604
15	31	393212
16	33	786428
17	35	1572860

Table 3

Number of nodes visited to execute Q_n on a D_n -conforming document.

and must account for more than just structural navigation. But do we see that, like the naive implementation, the comercial XPath implementation, `DTMNodeIterator`, also increased exponentially with n . So, clearly this is a case where a complete implementation of XPath could benefit from using strategies.

5.2 Applications

In this section we will examine some common applications (DTDs and queries) used in practice to try to predict a class of documents that should perform well with strategies. So, we first look at the previous example. Our XPath implementation with strategies out-performed the naive version because there was a large amount of the document that could never lead us to a successful results. In general, this is the desirable property of a document to look for in documents.

5.2.1 PIR-International Protein Sequence Database

This is an annotated protein database containing over 283,000 protein sequences. For our purposes it is very wide and shallow; that is, there many

```

<!ELEMENT ProteinDatabase (Database?,ProteinEntry+)>
<!ELEMENT Database (#PCDATA)>
<!ELEMENT ProteinEntry (header,protein,organism,
                        reference*,comment*,genetics*,
                        complex*,function*,
                        classification?,keywords?,
                        feature*,summary,sequence)>
                        ...
<!ELEMENT sequence      (#PCDATA)>
                        ...

```

Fig. 3. Abbreviated PIR DTD.

entries, but these entries are each not very large. An abbreviated version of the DTD for this database is shown in Figure 3 and the complete version is found in [10]. What's missing in this abbreviated version is that there are 59 elements before and nine after the `sequence` element. We single out the `sequence` element because a very common query is to do pattern matching on the sequence of every entry. Which means, that evaluation has to first navigate to all the `sequence` nodes and then do the pattern matching. With the naive approach either 59 or nine nodes are visited before arriving at the `sequence` node, where a meta-information based evaluation visits only node for each entry – the `sequence` node. Using meta information, the strategy graph is specified as `Root -> sequence`, so an evaluation based on this strategy graph leads directly from the `Root` node to `sequence` nodes.

5.2.2 *Biology as XML*

There are a number of projects that use XML and XML-like languages to represent protein and other biological concepts. The BIOpolymer Markup Language (BIOML) is an XML language that is used to describe experimental information about proteins, genes, and other biopolymers [5]; The Protein Sequence Database Markup Language is an open-standard markup language used to store protein information in the Protein Information Resource (PIR) database [11]. The Bioinformatic Sequence Markup Language is an open-standard protocol for the encoding and display of graphic genomic displays of DNA, RNA, and protein sequence information [4]. Genome Annotation Markup Elements is a markup language used in molecular biology for annotation of a biosequence [7]. The Multiple Sequence Alignments Markup Language was developed to make manipulation and extraction of multiple sequence alignment information easier by logically defining the parts of an alignment for use in an XML-based application [9]. We mention all of these to drive home the point that there is a growing trend to represent the natural world in XML, and this representation is often self-referential (i.e. recursive). When such a situation arises the efficient navigation becomes a must and using the meta information from DTDs can assist in this.

6 Discussion

What can XPath implementers learn from the Demeter experience? XPath expressions define possibly infinite sets of paths in the XML schema or DTD. The implementor needs to invent a suitable data structure to represent those possibly infinite path sets so that they can be used to guide the traversal of an XML document. We believe that the best way to represent the path sets is to use a so called traversal graph as described in [26] and [23].

It is important to notice that XPath is from one point of view more general than traversal strategies and from another point of view more limited than traversal strategies. XPath is more general because it allows predicates to select paths. In Demeter, such predicates are expressed using visitors and around methods. XPath is limited compared to traversal strategies because XPath only allows union of location paths and not general graphs. XPath is also limited because the semantics of an XPath expression with respect to an XML schema is an unordered node set without repetitions. The focus is only on the target nodes. In Demeter, the semantics is a traversal history that also cares about how the internal nodes are visited.

XPath uses a richer data model than Demeter class dictionaries. In XPath we have root nodes, element nodes, text nodes, attribute nodes, namespace nodes, processing instruction nodes, comment nodes while in Demeter we only have element nodes, some of them distinguished as root nodes. XPath requires that the objects are tree objects. The following definitions are used: Every node other than the root node has exactly one parent, which is either an element node or the root node. A root node or an element node is the parent of each of its child nodes. The descendants of a node are the children of the node and the descendants of the children of the node.

As a final note, the fragment of XPath considered only takes into account just structural information – i.e. *element* nodes. We do not consider attributes of these nodes. But, what we hoped to show in the previous section was that many practical, large databases are implemented using *element* nodes hold much of the information. But, as we note in Section 8, we could easily incorporate attributes when translating a DTD into a class graph so that our strategy would exploit these, too.

7 Related Work

Many approaches have been taken to efficiently execute queries over XML documents. Koch proposed a highly scalable and efficient evaluation scheme

based on tree automata [21]. Li and Moon use a numbering scheme of elements to store and index XML elements [22]. This numbering scheme quickly determines the ancestor-descendant relationship between elements in the hierarchy of XML data. structures and references. For example, XML metadata can be used to describe a web site structure to facilitate. Gottlob *et al.* presented various restricted fragments of XPath that lead to linear- and polynomial-time evaluation algorithms [20].

Amer-Yahia *et al.* presented seminal work on minimizing tree patterns queries (TPQs) in order to speed up evaluation [13]. Ramanan then improved on this approach by presenting more efficient algorithms for performing this minimization [29]. He reduced the previous $O(n^4)$ algorithm for minimizing TPQs in the absence of integrity constraints to $O(n^2)$ and the $O(n^6)$ algorithm for minimizing TPQs in the presence of only required-child and required-descendant constraints (this situation we consider) to $O(n^4)$. Flesca *et al.* show that, for larger fragments of XPath, evaluation is often NP-hard [19].

Very few people, though, have taken into account the meta information of XML documents. Chen *et al.* note that one can use relevant constraints from a document's schema to optimize a query [16]. They, then, show through empirical results that a navigation plan made using a document's schema can significantly outperform one made without such information. Bhowmick *et al.* show that DTD can be exploited when storing XML in databases [14]. This is different than our approach, because this project *stores* XML documents in databases rather than implementing databased using XML. But, it, nevertheless, shows that the meta information of XML documents is important when for efficient implementation of queries.

8 Conclusions and Future Work

We have shown that there is a close relationship between object graphs and XML documents. Likewise, there is a close relationship between the meta data describing object graphs – class graphs – and the meta data describing XML documents – DTDs. Noting this relationship we were first able to show that ignoring an XML document's DTD when evaluating an XPath query could cause this query to visit exponentially many nodes; while, using this meta data could we could visit a linear number of nodes. We, finally, showed that this work is meaningful because many large databases are mainly structural, and therefor benefit from this speed up.

Future work will include first a look at a larger fragment of XPath so that we can create a real-world implementation based on strategies. In doing this we will have to consider attributes and more of the structural query features.

After having a full or close-to-full implementation of XPath, we can do extensive empirical testing to show that it is, in fact, the case that strategies are beneficial for very large databases implemented in XML in practice.

9 Acknowledgments

TODO

References

- [1] Xml schema. <http://www.w3.org/XML/Schema>.
- [2] Pir-international protein sequence database. ftp://nbrfa.georgetown.edu/pir/databases/pir_xml/, November 2004.
- [3] Ap library. <http://www.ccs.neu.edu/research/demeter/software/APLibrary/>, 2005.
- [4] Bioinformatic sequence markup language. <http://www.labbook.com/>, 2005.
- [5] Biopolymer markup language (bioml). <http://www.bioml.com/BIOML/index.html>, 2005.
- [6] Demeterj. <http://www.ccs.neu.edu/research/demeter/software/DemeterJ/>, 2005.
- [7] Genome annotation markup elements. <http://www.bioxml.org/Projects/game>, 2005.
- [8] Java 2 platform standard edition 1.5. <http://java.sun.com/j2se/1.5/>, 2005.
- [9] Multiple sequence alignments markup language. <http://maggie.cbr.nrc.ca/gordonp/xml/MSAML>, 2005.
- [10] Pir database dtd. ftp://ftp.pir.georgetown.edu/pir_databases/psd/xml/, 2005.
- [11] Protein sequence database markup language. <http://pir.georgetown.edu/>, 2005.
- [12] Xerces-j api documentation. <http://xml.apache.org/xerces-j/apiDocs>, 2005.
- [13] S. Amer-Yahia, S. Cho, L. V. S. Lakshmanan, and D. Srivastava. Minimization of tree pattern queries. In *Proceeding of the ACM SIGMOD international conference on the management of data*, pages 497–508. ACM Press, May 2001.
- [14] S. S. Bhowmick, T. K. Wee, E. Leonardi, and S. K. Madria. Storing dtd-conscious xml data in xedy. In *EC-Web*, pages 270–280, 2003.

- [15] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, and J. R. J. Simeon. Xquery: An xml query language. <http://www.w3.org/TR/xquery/>.
- [16] Z. Chen, H. V. Jagadish, L. V. S. Laksmanan, and S. Pappas. From tree patterns to generalized tree patterns: On efficient evaluation of xquery. In *VLDB '03: Proceedings of the 29th Intl. Conf. on Very Large Data Bases*. ACM Press, 2003.
- [17] S. DeRose, E. Maler, and R. D. Jr. Xml pointer language. <http://www.w3.org/TR/WD-xptr>.
- [18] S. DeRose, E. Maler, and D. Orchard. Xml linking language (xlink) version 1.0. <http://www.w3.org/TR/xlink>, 2001.
- [19] S. Flesca, F. Furfaro, and E. Masciari. On the minimization of xpath queries. In *VLDB*, pages 153–164, 2003.
- [20] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing xpath queries. In *In Proceedings of the 28th International Conference on Very Large Data Bases (VLDB'02)*, 2002.
- [21] C. Koch. Efficient processing of expressive node-selecting queries on xml data in secondary storage: A tree automata-based approach. In *VLDB*, pages 249–260, 2003.
- [22] Q. Li and B. Moon. Indexing and querying XML data for regular path expressions. In *The VLDB Journal*, pages 361–370, 2001.
- [23] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.
- [24] K. Lieberherr and M. Wand. Navigating through object graphs using local meta-information. Technical Report NU-CCS-2001-05, Northeastern University, May 2001. <http://www.ccs.neu.edu/research/demeter/biblio/new-strategy-semantics.html>.
- [25] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X, entire book at www.ccs.neu.edu/research/demeter.
- [26] K. J. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of Object Structures: Specification and Efficient Implementation. *ACM Transactions on Programming Languages and Systems*, March 2004.
- [27] J. Marshall, D. Orleans, and K. Lieberherr. DJ: Dynamic Structure-Shy Traversal in Pure Java. Technical report, Northeastern University, May 1999. <http://www.ccs.neu.edu/research/demeter/DJ/>.
- [28] G. Miklau and D. Suciu. Containment and equivalence for an xpath fragment. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 65–76. ACM Press, 2002.

- [29] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 299–309. ACM Press, 2002.
- [30] P. Wadler. A formal semantics of patterns in xslt, 1999.
- [31] P. Wood. Containment for xpath fragments under dtd constraints, 2003.