Approximate Retrieval of XML Data with ApproXPath

Lin Xu[1] **Curtis Dyreson**[2]

[1] IBM Santa Teresa
Santa Teresa, CA
Email: linxu@ibm.com

[2] Department of Computer Science
Utah State University
Logan, UT
Email: Curtis.Dyreson@usu.edu

Abstract

Several XML query languages have been proposed that use XPath expressions to locate data. But XPath expressions might miss some data because of irregularities in the data and schema of an XML data collection. In this paper we propose ApproXPath, which supports approximate path expressions. Approximate path expressions have the same syntax as XPath expressions, but allow content and structural errors. An error is a string or tree edit operation that creates a (virtual) data collection in which the data can be located. ApproXPath extends XPath's axes, node tests and predicates to utilize the string/tree edit distance. We show that the complexity of ApproXPath is reasonable. For many queries, the inexact matching (with no errors) is as fast as exact matching, and the cost increases linearly with the number of errors allowed.

1 Introduction

XML has become an important language for representing and exchanging data. Part of the appeal of XML is that it is flexible. XML allows small, local differences in the structure of data. For instance, in a collection of book data, there may be an editor for a book, or no editor, or several editors enclosed within an <editors> element. But the flexibility of XML in representing data can make it more difficult to retrieve data. Several XML query languages have been proposed that use XPath expressions to locate data, most notably, XQuery. These query languages use XPath because it is a simple, easy-to-use language for specifying a data location. However, when small, local differences in the structure of the data are unanticipated or not known a priori, crafting an XPath expression to locate data becomes more challenging.

This paper proposes ApproXPath which supports *approximate* path expressions. Approximate path expressions can handle small structural differences when locating data. Approximate path expressions have the same syntax as XPath, but are evaluated differently. The evaluation of an ApproXPath expression allows a user-specified number of content and structural errors. An error is a string or tree edit operation that creates a (virtual) data collection in which the data can be located. The number of errors allowed corresponds to an edit distance. If n errors are allowed, nodes that are in some trees that are n edits away from the original data are included in the results (tagged with the distance).

We had several design goals for ApproXPath, which are common goals in *query relaxation* systems (1).

- The result with zero errors should be identical to the conventional XPath result.
- The set of approximate matches to a query should be a super set of the exact matches.
- The language should have a set of scoring mechanisms to specify the error bound.
- Exact matches for a query must have the highest score.
- The score can be used to rank the results in relevance order (i.e., no errors has a higher rank than three errors).

Additionally we wanted to utilize *agrep*, a well-known approximate string matching tool (2; 3), and focus exclusively on XPath, a path expression language that is a small part of many other languages and systems promulgated by the W3C.

As an example consider the following XPath query which is designed to locate the title of each book published by Zeus.

```
//book/title[../publisher="Zeus"]
```

As this query is executed, it can be relaxed into the following queries that allow one "structural" error (a node insertion or deletion on a path).

```
//book/*/title[../publisher="Zeus"]
//book/title[../..//publisher="Zeus"]
//book/title[../*/publisher="Zeus"]
```

Or one error in "content" by using an approximate string matching algorithm to match any one of the labels on the path (i.e., <title> would match <tilde>). The nodes in the result would be ranked by the number of errors.

The contributions of this paper are listed below.

- Adds approximate axes and node tests to XPath.
- Query results are sound (for uni-directional queries, see Section 4) in the sense that nodes in the result are within a specified edit distance.
- Irrelevant results are eliminated as soon as possible to achieve the optimal speed. Our analysis suggests that ApproXPath will have reasonable performance.

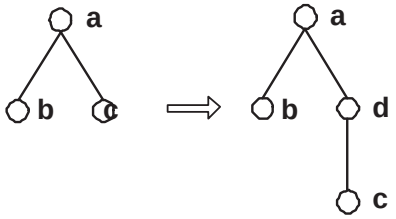

Figure 1: Node insertion

The remainder of the paper is organized as follows. In Section 2 we review the tree-like model of XML and tree edits. The details of ApproXPath are described next followed by a description of the ApproXPath evaluation algorithm. We implemented ApproXPath and report on some experiments using a Java prototype in Section 5. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 Tree Edit Operations

An XML document, D, can be modeled (with some simplifications) as an ordered tree.

Definition 2.1 (Ordered, Labeled Tree) *A tree is a structure $T = (V, E, L)$ where*

- *V is a (finite) set of nodes,*
- *E is a set of edges such that $E \subseteq V \times V$ where E has no cycles and every node is connected,*
- *for any $v, x \in V$, if $\exists p \in V$ such that $(p, x) \in E$, $(p, v) \in E$, and v lexically appears before x in D, then $v < x$ in the sibling order,*
- *there is a single node $r \in V$ that is called the root, and*
- *L is the set of labels and one label is associated with each node.*

In a data model instance of an XML document, each interior node in the tree corresponds to an element, comment, or processing instruction. Each leaf is a (text) value or attribute (a name, value pair). Finally, the children for a node are ordered corresponding to their lexical order in the document (though the order of attributes does not matter).

2.1 Edit Operations on a Tree

There are four tree edit operations: context change, node delete, node insert, and subtree swap, in a complete set of such operations (4; 5; 6).

The node insert operation takes an edge (a, c) and a node d and inserts d as a child of a and parent of c, as shown in Figure 1.

Definition 2.2 (Node Insertion) *Let $T = (V, E, L)$ be a labeled tree. The insertion of node $v \notin V$ on edge $(u, w) \in E$ is a function such that $insert(T, v, (u, w)) = (V', E', \text{label}(v) \cup L)$ where $V' = V \cup \{v\}$, and $E' = E - \{(u, w)\} \cup \{(u, v), (v, w)\}$*

The delete operation for node c removes node c and makes its children the children of its parent, as shown in Figure 2.

Definition 2.3 (Node Deletion) *Let $T = (V, E, L)$ be a labeled tree. The deletion of node $v \in V$ from T is a function of T such that $delete(T) = (V', E', L - \text{label}(v))$ where $V' = V - \{v\}$, and $E' = E - (\{(x, y) \mid (x, v) \in E \vee (v, y) \in E\} \cup \{(u, w) | (u, v) \in E \wedge (v, w) \in E\})$*

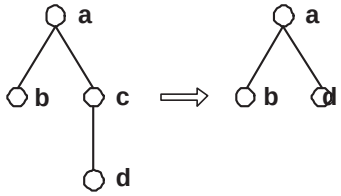

Figure 2: Node deletion

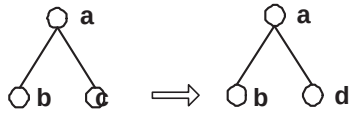

Figure 3: Node relabeling

The context change operation on node c changes the label of the node, as shown in Figure 3.

Definition 2.4 (Node Relabeling) *Let $T = (V, E, L)$ be a XML tree and $v \in T$ be a node. A relabeling of v changes its label from x to y. that is, $relabel(T, v, y) = (V, E, L')$ where if $\forall u \in V$:*

$$label'(u) = \begin{cases} y & \text{if } u = v \\ label(u) & \text{otherwise} \end{cases} \quad (1)$$

Subtree swap swaps siblings as shown in Figure 4.

Definition 2.5 (Subtree Swap) *Let $T = (V, E, L)$ be a labeled tree. The subtree swap of nodes $u, v \in V$ that are children of some $p \in V$ swaps u and v in the node ordering. For any nodes $x, y \in (V - u, v)$ if $x < v$ in T and $y < u$ in T then $x < u$ in the result and $y < v$ in the result.*

An arbitrary nonnegative cost function *cost*() is associated with each type of the edit operation. The cost function has the following characteristics.

- $cost() \geq 0$
- $cost(T \rightarrow T) = 0$
- $cost(T \rightarrow T') = cost(T' \rightarrow T)$
- $cost(T \rightarrow T') + cost(T' \rightarrow T'') \geq cost(T \rightarrow T'')$

The above is easily satisfied if each kind of operation has the same nonnegative cost.

Definition 2.6 (Cost of Edit Operations) *The cost of a sequence of edit operations $S = e_1, e_2, ..., e_n$ is defined as $\sum_{i=1}^{n} cost(e_i)$.*

An important concept is the minimal edit distance.

Definition 2.7 (Minimal Edit Distance) *The minimal edit distance between two trees T, T' is defined by the minimal cost of the sequence of edit operations that transforms one tree to the other.*

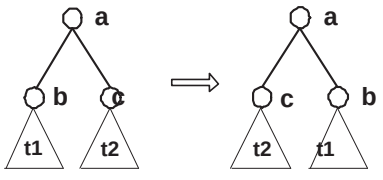

Figure 4: Subtree swapping

$dist(T, T') = min\{\sum_{i=0}^{m} cost(S_i) \mid S_i$ *is a sequence of edit operations,* $e_1, e_2, ..., e_n$*, that changes* T *to* T'*, i.e.* $T' = e_m(...(e_2(e_1(T)))\}$

The distance metric satisfies the triangle inequality: $dist(T_1, T_3) \leq dist(T_1, T_2) + dist(T_2, T_3)$.

3 ApproXPath

This section discusses ApproXPath, which has the same syntax as conventional XPath but relaxes its semantics to locate nodes that are within the specified number of errors. We first review XPath and then present ApproXPath.

3.1 Review of XPath

The primary purpose of XPath is to address parts of an XML document. Formally, an XPath expression is a sequence of *location steps*, $s_1, s_2, \ldots, s_m$ which are evaluated in sequence. Let $E_X(T, P, c)$ represent the evaluation of an XPath expression, P, on data model instance, T, starting from a (set of) context node(s) c.

$$E_X(T, P, c) = E_X(T, s_m, \\ E_X(T, s_{m-1}, \ \ldots \\ E_X(T, s_1, c) \ \ldots \ ))$$

A step consists of an *axis*, a *node test*, and zero or more *predicates*. The axis specifies a list of nodes located in some portion of the tree relative to the context node. A common axis is *descendant* which locates all of the descendants of the context node. The node test is then applied to each node in the axis. Nodes that pass the node test are further tested by each predicate. If each predicate is satisfied, the node is placed into the output of the step. Each step takes the set of context nodes produced by the previous step as input, and produces a node-set. The result of the last step is the result of the expression.

3.2 ApproXPath

ApproXPath extends XPath by allowing a certain number of errors in locating nodes. Given an XML document, an error bound n, and an XPath location expression e, the result of an ApproXPath expression is the nodes that are in the original data model tree T and those that are in some tree T' that is within n edits from tree T. More specifically, we have the following definition.

Definition 3.1 (Result Set of ApproXPath) *Given XML tree,* T*, an error bound,* n*, and an XPath expression,* P*, the result set of evaluating* P *with ApproXPath semantics,* $E_A(T, P, c, n)$*, from the set of context nodes* c *is*

$$E_A(T, e, c, n) = \{x \mid \exists T'[x \in E_X(T', P, c) \\ \wedge \ x \in node(T) \\ \wedge \ dist(T', T) \leq n]\}$$

where $E_X(T, P, c)$ *denotes the conventional XPath expression* P *applied on* T *from the set of context nodes* c*.*

Two things need to be clarified in the definition.

1. It is not required that tree T' be the same for each element in result.
2. Nodes in the result come only from the original tree T. Though ApproXPath allows the logical insertion of spurious nodes in the original tree, no node that was inserted is in the result.

ApproXPath uses *query relaxation* to approximately locate nodes. Query relaxation relaxes the specification and semantics of an expression in two ways: structural relaxation and content relaxation. Structure relaxation relaxes the relationship between two nodes in the tree. For example, the query `/book/title` finds only <`title`> nodes that are children of <`book`> nodes. The expression can be structurally relaxed by modifying the semantics of the child axis to include both the children and the grandchildren of the context node (corresponding to a delete edit of the tree). Content relaxation on the other hand would involve modifying the label to search <`titles`>, <`titled`>, <`books`>, etc.

ApproXPath replaces XPath's axis and node test with an *InexactAxis* that allows structural errors and *InexactNodeTest* and *InexactPredicate* that allow content errors.

3.2.1 *InexactAxis*

The inexact axes for zero errors are unchanged from XPath. For one or more errors, they are listed below.

- The following axes have no inexact match. No sequence of inserts, deletes, subtree swaps, or node relabeling will change their result. (Recall that spurious nodes, that is, nodes not in the original tree introduced by a node insert operation are not in a result.) In some sense, these axes already do "approximate" location searches in a tree.

  $$\texttt{ancestor}_{inexact}(k) \rightarrow \emptyset$$
  $$\texttt{ancestor-or-self}_{inexact}(k) \rightarrow \emptyset$$
  $$\texttt{descendant}_{inexact}(k) \rightarrow \emptyset$$
  $$\texttt{descendant-or-self}_{inexact}(k) \rightarrow \emptyset$$
  $$\texttt{self}_{inexact}(k) \rightarrow \emptyset$$

- The `attribute` axis has only a one error match, essentially converting an attribute into a subelement.

  $$\texttt{attribute}_{inexact}(1) \rightarrow \texttt{child}.$$

  This corresponds to a label change of the XML tree. The type of the node is changed from attribute to other non-attribute type. The cost of this change is 1.

- *child* has several inexact matches. Inexact match with k errors is

  $$\texttt{child}_{inexact}(k) \rightarrow \underbrace{\texttt{child/.../child}}_{k+1}.$$

  This corresponds to a series of (spurious) insertions/deletions to the XML tree.

- The subtree swap operation permutes the order of siblings. Just one such operation will change the order, so axes that depend on the order have one error match.

  $$\texttt{following}_{inexact}(k) \rightarrow \texttt{preceding}$$
  $$\texttt{following-sibling}_{inexact}(k) \rightarrow \texttt{preceding-sibling}$$
  $$\texttt{preceding}_{inexact}(k) \rightarrow \texttt{following}$$
  $$\texttt{preceding-sibling}_{inexact}(k) \rightarrow \texttt{following-sibling}$$

- Like the `child` axis, the `parent` has k errors inexact match.

$$\texttt{parent}_{inexact}(k) \rightarrow \underbrace{\texttt{parent/.../parent}}_{k+1}.$$

This corresponds to a series of (spurious) insertions/deletions to the XML tree.

If we take a closer look at those axes that have inexact match, such as the `child` axis, we observe that there in obtaining a result with error n, we already compute the result with $n-1$ errors as a byproduct. Thus, the result of each *InexactAxis* is an array of node sets. Each node set in the array represents a result that corresponds to a specified error.

3.2.2 *InexactNodeTest* and *InexactPredicate*

Nodes that are in the axis must still pass a node test to remain in the result. Though there are several kinds of node tests, the most common is a name match, wherein the name of a node is string-matched against a name in the query (e.g., in a test to ensure that <`title`> nodes are found). An error in name match is a content error. Fortunately, many excellent approximate string matching techniques exist. We adopted the *agrep* algorithm developed at the University of Arizona into ApproXPath. *agrep* is an approximate *grep* that supports string matching with k string-edit errors (2; 3).

Though trivially we could treat string-edit and tree-edit errors as the same (i.e., allowing k errors in a result would mean either allowing k string-edit operations or k tree-edit operations, or any combination thereof which counted to k) we decided that the two kinds of errors were not equivalent. So we set a threshold for the content errors (mismatched characters) in the name match. Since the length of the name can vary greatly, the threshold is a percentage of the string length of the node name, The threshold is adjustable in the implementation, and initially we choose 40% in our ApproXPath. So for a name of length 10, it will approximately match with at most 4 string-edit operations. We count the match as 1 error overall.

In summary, an *InexactNodeTest* can have one of the three kinds of results: successful with no error, successful with one error, or failure. For the other kinds of node tests, *nodetype*() and *processing-instruction*(), we did not introduce an inexact semantics as they had no natural tree edit analog.

agrep was also introduced for *InexactPredicate*s. In general, there could be several predicates in a step. The predicates are evaluated from left to right. If a predicate contains a LocationPath, then the LocationPath is evaluated under the ApproXPath semantics using the *InexactAxis*s and *InexactNodeTest*s. Errors found in a predicate add to the overall error. Some predicates use an equality comparison to compare a sibling position (pos(), etc) to a number. One error is allowed here (which matches any position) corresponding to a subtree swap. The equality comparison is also used to compare two strings. We use *agrep* as in the node match for these comparisons, allowing one error.

3.2.3 Putting it all Together

The result of ApproXPath is the result of *InexactAxis*, *InexactNodeTest* and *InexactPredicate* for each location step. The errors are accumulated, resulting in a list of nodes with each node having an accumulated error.

The FSM shows that nodes in the intermediate result set with k errors are related to nodes with $k+i$ errors if i errors are introduced during the next evaluation step. If $k+i > n$, where n is the total number allowed, the result is discarded.

4 ApproXPath Evaluation Plan

This section presents a navigation-based approach to ApproXPath evaluation where queries are evaluated step-by-step which is common in many in-memory XPath query engines such as Apache Xalan. Optimizing this approach through the use of indexes is beyond the scope of this paper.

The navigation-based approach to ApproXPath evaluation has three overall concepts.

1. The algorithm builds a list of node objects in the result, R, sorted in document order. Each node object contains a node and an error count. A node can be added to the list only when the node is missing. A node can be replaced in the list only when the error count is less than the current node. In other words, if we currently have the node x in the result with an error of k we would replace x only if the error were less than k.

2. For each step in a query perform the following, an appropriate *InexactAxis*, *InexactNodeTest* and *InexactPredicate* is applied to each node in the current result, R, yielding a new list R, which forms the starting point for the next step.

This evaluation strategy is "memory-less" since we do not keep track of the tree edits to reach a particular node. We adopt a memory-less approach for reasons of efficiency. Keeping track of the edits would make the cost exponential. Suppose that in the result we have a node x with j errors.[1] The query to this point has explored some tree T_j at an edit distance of j from the original tree, T_0, but which T_j? It could be a T_j with i insertions and m deletions ($i+m=j$) or a tree with m deletions and i insertions. There are an exponential number of possible T_j's. If a node is matched in the n^{th} step of ApproXPath with error j there are

$$\begin{pmatrix} n+j-1 \\ j \end{pmatrix}$$

trees at distance j from the original tree.

The disadvantage of a memory-less approach is that it may miscalculate the error. We could overcount errors when backward and forward directional axes are mixed e.g., a query that uses a child axis and a parent axis mixes directions. When directions are mixed a (spurious) node that was inserted to reach a node may be re-inserted when the query goes backwards, counting the insertion twice. But if an XPath location path contains only forward or only backward directional axes, then we can claim the following about memory-less ApproXPath.

Claim: If an XPath expression consists of only forwards or backwards axes, and navigation-based ApproXPath evaluation is used, then for every node in the result, there exists a tree T' that contains the node such that $dist(T,T') \leq \sum_{i=1}^{m} d_i$, and $T' \cap T_i = T_i$, where m is the number of evaluation steps, and $d_1, d_2, \ldots d_m$ are the edit distance introduced in evaluation step $1, 2, \ldots m$.

[1] x is always a node in the original XML tree, T_0, that is, it is not a "spurious" node introduced as an insertion; the result is always limited to nodes in the original tree.

An algorithm for navigation-based evaluation is shown in Figure 5. The inputs of the navigation algorithm are an XML document tree T, an XPath expression e and the error bound n. The return value of the algorithm is a list of nodes.

Analyzing the complexity is straightforward. Let m be the number of steps in an XPath expression, p be the number of predicates in any one step, k be the error allowed, and n be the total number of nodes in the tree. Then each node is visited at most $O(k\ p\ m)$ times. For each node visited, the cost to merge the node in the node list is at most $O(\log_2 n)$ (using binary search). So the overall cost of the algorithm is $O(k\ p\ m\ n\ \log_2(n))$. For the zero error case, the cost is $O(p\ m\ n\ \log_2(n))$, which is the cost of (naive) XPath evaluation. So ApproXPath adds an overhead of a factor of k, that is, the cost grows linearly in the number of errors allowed. Note that k, p, and m are typically much smaller than n.

The space complexity is modest. The maximum size of a node list can not exceed the size of the tree so the space complexity is $O(n)$.

5 Implementation

We implemented ApproXPath in Java[2] by modifying Apache Xalan[3]. We utilized Xalan's parser, but re-implemented (part of) the back-end using the algorithm described in Section 4, which is a modification of Xalan's evaluation algorithm. We took this approach to demonstrate that our algorithm can adapt to an existing XPath implementation, and to make the comparison with Apache Xalan fairer because we share the front end and part of the back end.

We tested ApproXPath on randomly generated trees from an XPath benchmark (7). The trees varied on the following factors.

- Degree of the document root - This factor represents the number of the children of the document root. It controls the top level bushiness of the tree.
- Depth - This factor represents the nesting of elements in the XML document. It controls the depth of the XML tree.
- Bushiness - This factor describes the number of children (degree) in a non-leaf node in the XML tree. The bushiness can be fixed or chosen randomly from a range.
- Attributes - The number of attributes in the leaf nodes. We fixed this at 10.

We performed tests on three different kinds of trees. Each kind has a different shape.

- Bushy - Documents of 9MB, root children 5, bushiness of 5, depth 6
- Tall - Documents of 0.3MB, root children 5, depth 500, bushiness 1
- Fat - Documents of 5.73 MB, root children 100, depth 2, bushiness 1

For each test, we amortized results over 10 runs.

We performed some tests to establish a base-line for our modified Xalan engine. We compared the cost of running a query on Xalan with running the same query on ApproXPath with no errors. The query mixes we used in the tests are given in Appendix A. The results are shown in Figures 6, 7, and 8.

[2]Sun J2SE 1.4.2_03

[3]Apache Xalan J-2.5.1 XPath package

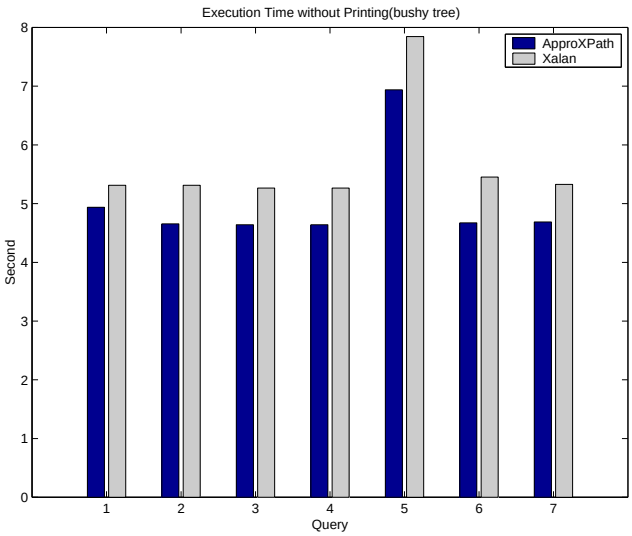

Figure 6: Base-line execution time on bushy trees

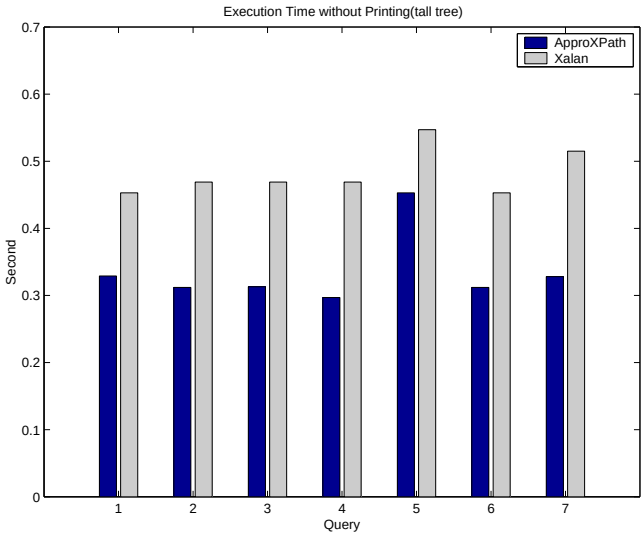

Figure 7: Base-line execution time on tall trees

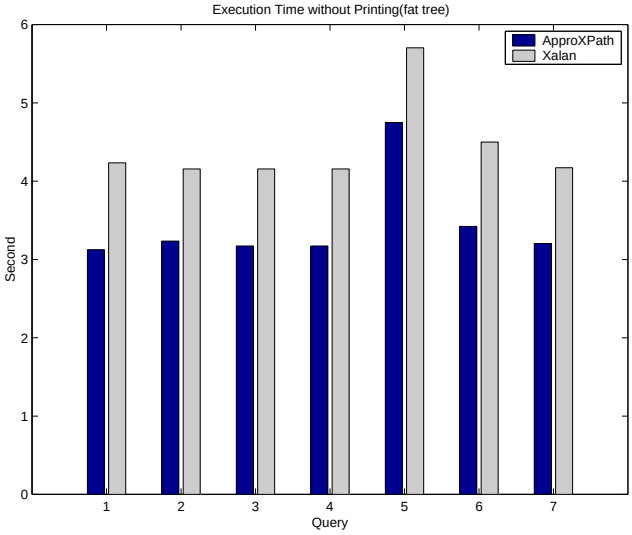

Figure 8: Execution time on fat trees

```
NodeList InexactAxis(R:node list, a:axis, n:max error allowed)
    1.  NodeList S = ()
    2.  foreach node c ∈ R
    3.      S = merge(S, result of applying inexact axis a on context node c)
    4.  return S

NodeList InexactNodeTest( c:nodes, t:nodeTest)
    1.  NodeList S = result of applying inexact node test t to each context node in c
    2.  return S

NodeList approPredicate (R:context nodes, P:list of predicates, n:maximum error permitted)
    1.  NodeList X = ()
    2.  foreach c ∈ R
    3.      foreach (predicate p ∈ P)
    4.          foreach LocationPath e in p
    5.              NodeList S = approLocationPath (e, n, R)
    6.              R = S
    7.          if S is nonempty then add c to X
    8.  return X

approLocationPath (e:LoactionPath, n: maximum error permitted, R:input/output NodeList)
    1.  Divide the LocationPath e into m location steps, e = s_1, s_2, ..., s_m
    2.  NodeList R = ()
    3.  for i = 1 to m
    4.      NodeList S = InexactAxis(R, s_i.axis, n − j)
    5.      S = InexactNodeTest(S, s_i.nodetest)
    6.      R = approPredicate(S, p, n − j)
```

Figure 5: An Algorithm for the Navigation-based Evaluation

These results suggest that ApproXPath with zero errors performs about the same as Xalan, which is what we expected. ApproXPath is slightly faster due to minor differences in the back-end, but this speed-up is of no importance.

Next, we performed several tests to determine how the cost increases as the number of errors allowed increases. The complexity analysis in Section 4 suggested that the increase is (at worst) linear in the number of errors allowed. We tested various queries on the three kinds of trees listed above: bushy, tall, and fat. For each query we graphed the overall execution time (with and without printing the nodes) and also the size of the result (in number of nodes). Figures 9 through 20 show some of our results.

These tests are by no means exhaustive or comprehensive, rather they are very preliminary. But from this preliminary data, two clear trends emerge.

1. As more errors are allowed, the query result becomes meaningless since the result expands to include the entire tree. Sometimes this happens quite rapidly (at two errors in Figure 11). To some extent this is an artifact of the kinds of tests we ran. All of our tests involve queries that are "range" queries rather than point queries. A point query is a query that pinpoints a node or small group of nodes in a tree that satisfy some constraints. A range query on the other hand returns a large set of results. We chose to experiment with range queries because they cover large portions of the tree, involve large lists of intermediate results, and query evaluation times are largely a function of the size of the result. Observe that the "jump" in query execution time in Figure 9 occurs when the errors allowed reaches 5, which correspond to the "jump" in the size of the result at 5 errors. In general, a query relaxation system will quickly generate more data than users can handle. To mitigate this, many such systems suggest ranking query results and presenting only the "top-k" to users.
2. The cost pretty much grows linearly as more errors are allowed, and tends to flatten out when the maximum result size is reached, i.e., when all the nodes in the tree are in the result.

6 Related Work

There are two primary areas of related research: distance metrics in editing strings or trees and approximate query languages for XML.

Approximate string matching introduced the idea of using edit distances (8; 9; 2; 3). This is still an active research area with recent applications in biological sequence matching and web search. (10) provides a very good survey. We were inspired by Wu and Manber's work on *agrep* (2; 3).

Tai's paper (4) is the first to address edit distances in trees. But more recent papers cover a range of tree matching problems (5; 11; 6). including a proof that unordered tree embedding is NP-Hard (11). ApproXPath adapts the tree-matching research to query evaluation.

ApproXPath is more related to approximate answering of tree pattern queries, which in turn is closely related to approximate keyword matching in IR (12). There has been significant research in IR on indexing techniques and query evaluation heuristics that improve the query response time while maintaining a constant level of relevance to the initial query (c.f., (13; 14; 15; 16; 17; 18)). In fact, our strategy of using data pruning for obtaining answers to relaxed tree patterns was inspired by the work done in IR. However, our evaluation and optimization techniques differ from the IR work, because of our emphasis on tree-structured XML documents.

There are several language proposals for approximate query matching. These proposals can be classified into two main categories: approximate content and approximate hierarchies. In the first category are approximate text searches in documents (using predicates such as "near") (19; 20; 21; 22). The second category adds approximate structural matches.

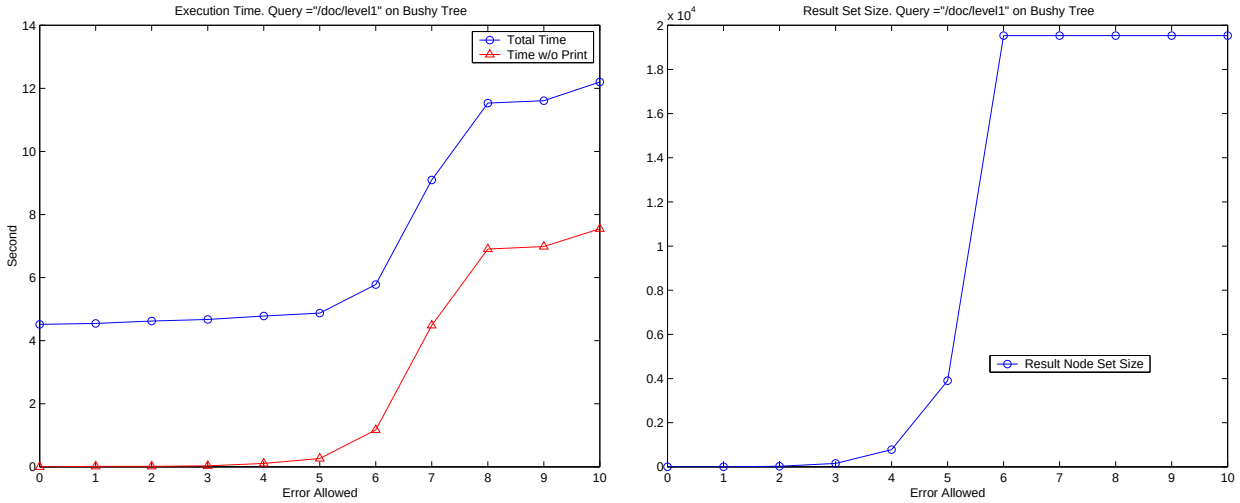

Figure 9: Testing query "/doc/level1" on bushy trees

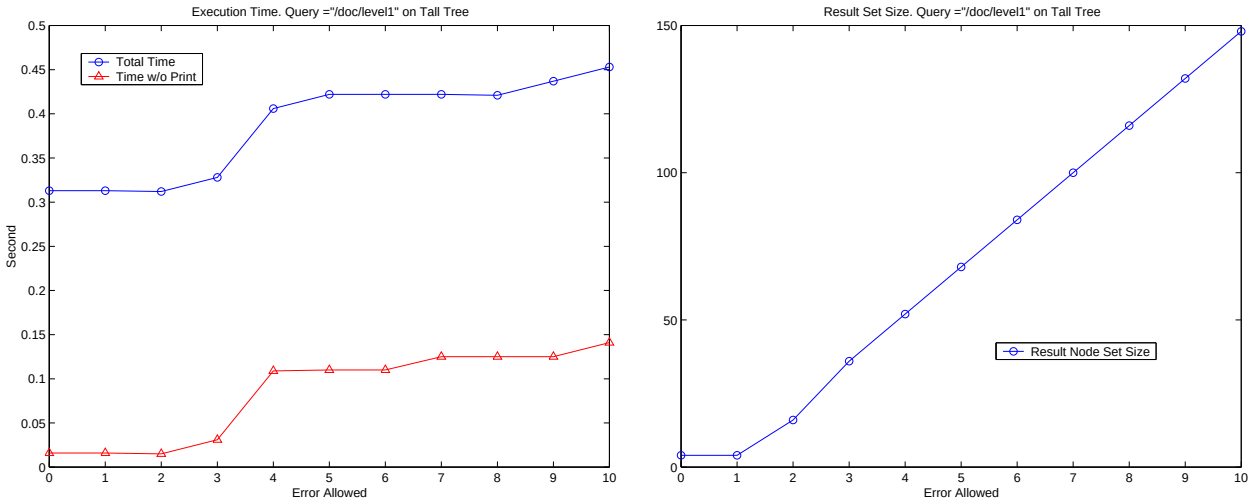

Figure 10: Testing query "/doc/level1" on tall trees

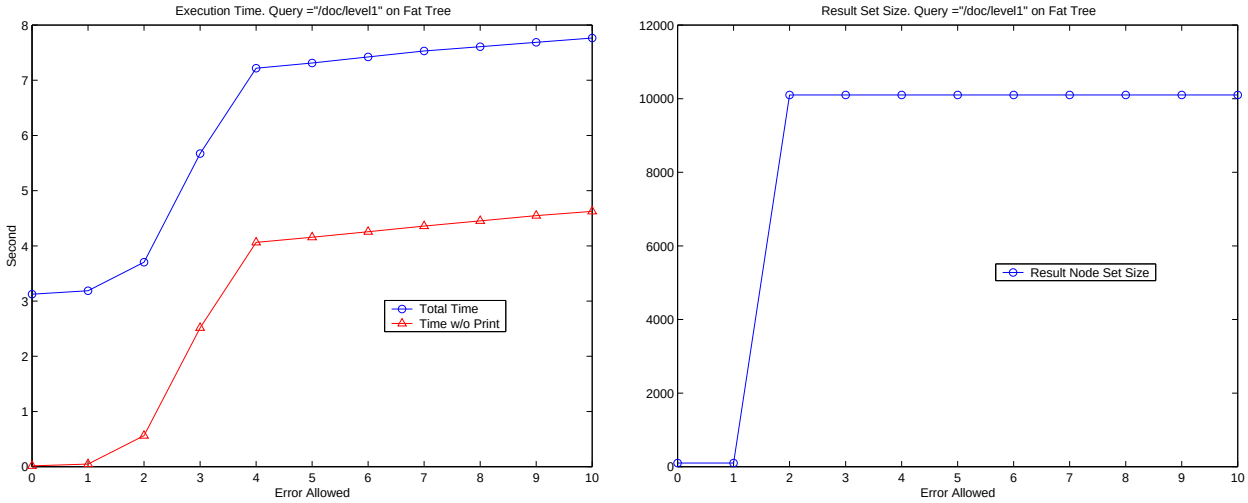

Figure 11: Testing query "/doc/level1" on fat trees

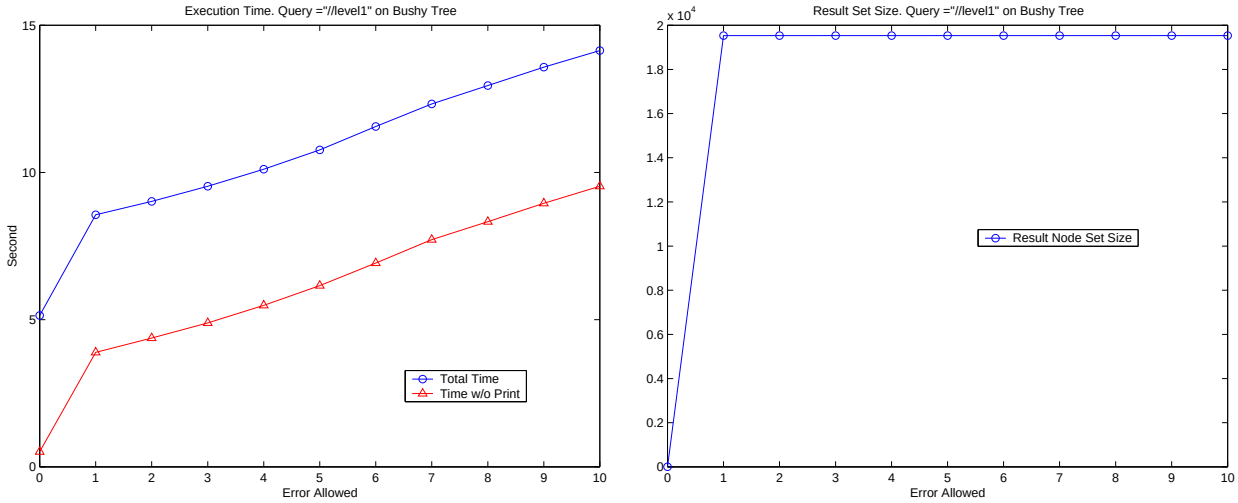

Figure 12: Testing query "//level1" on bushy trees

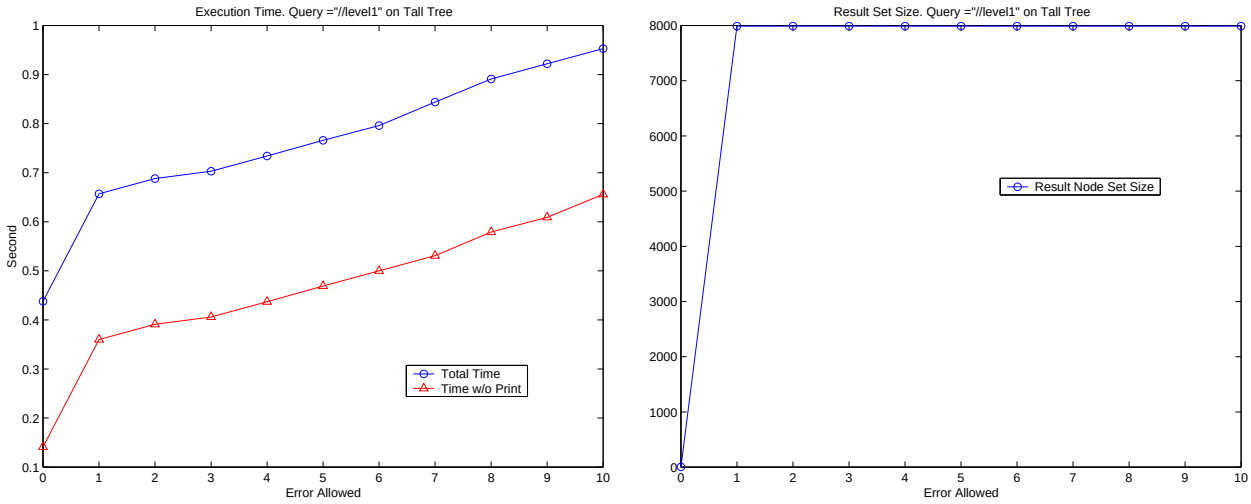

Figure 13: Testing query "//level1" on tall trees

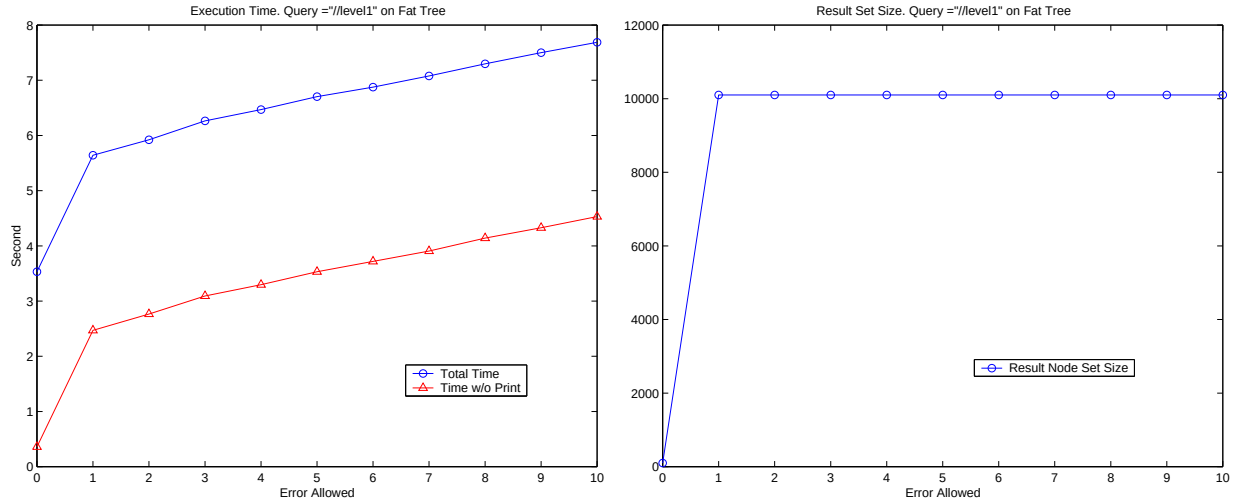

Figure 14: Testing query "//level1" on fat trees

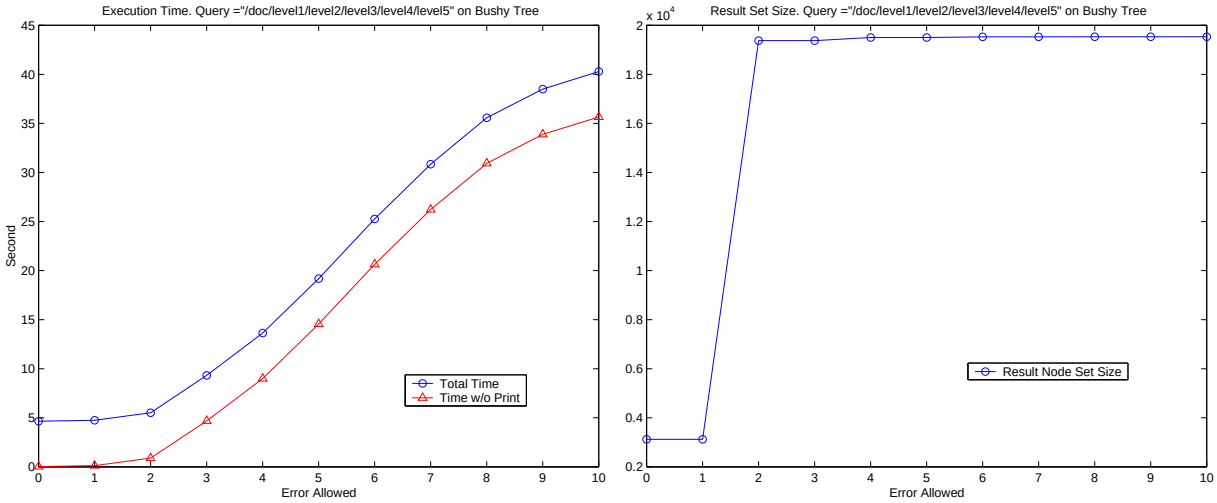

Figure 15: Testing query "/doc/level1/level2/level3/level4/level5" on bushy trees

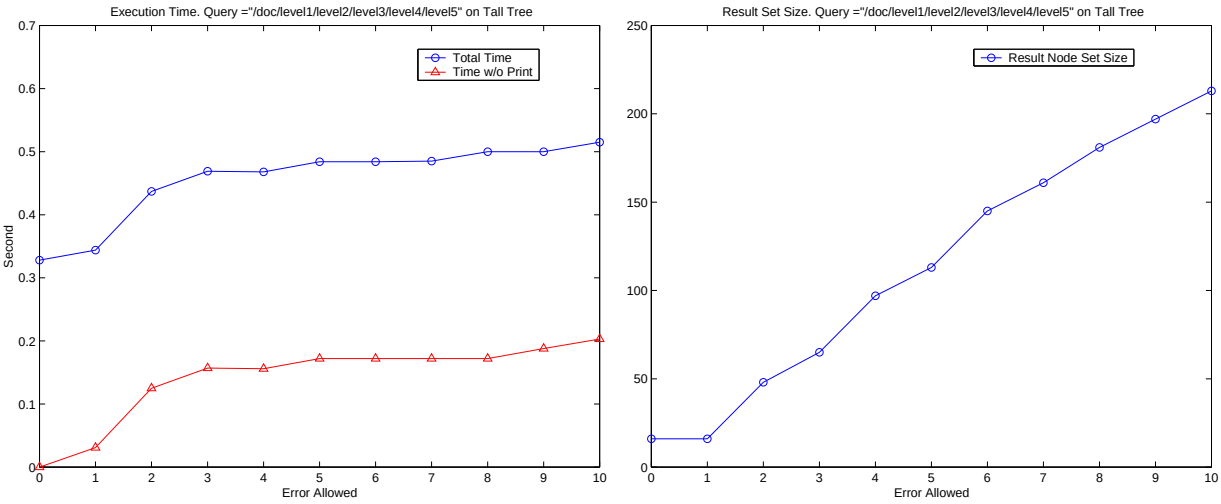

Figure 16: Testing query "/doc/level1/level2/level3/level4/level5" on tall trees

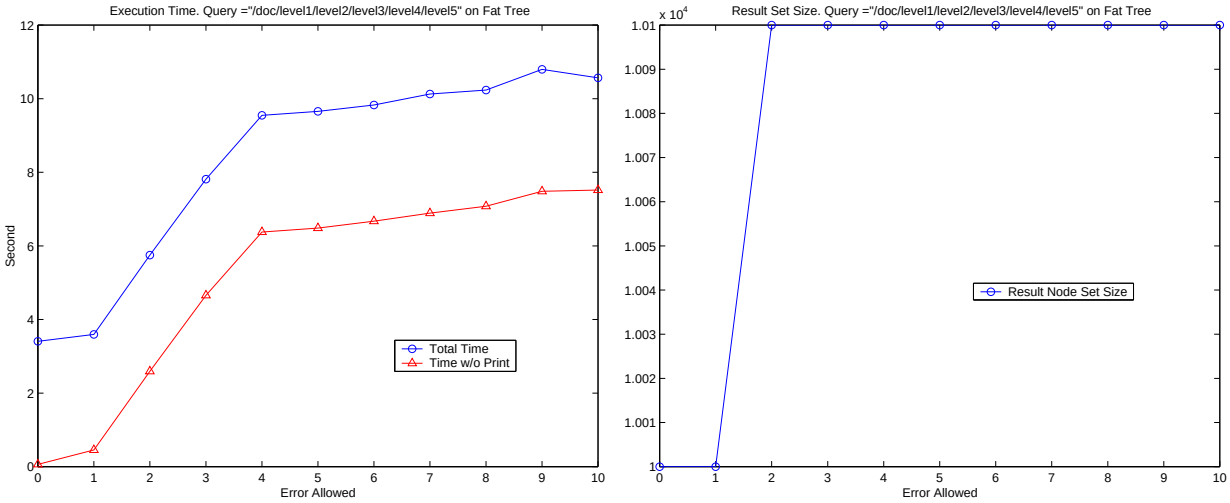

Figure 17: Testing query "/doc/level1/level2/level3/level4/level5" on fat trees

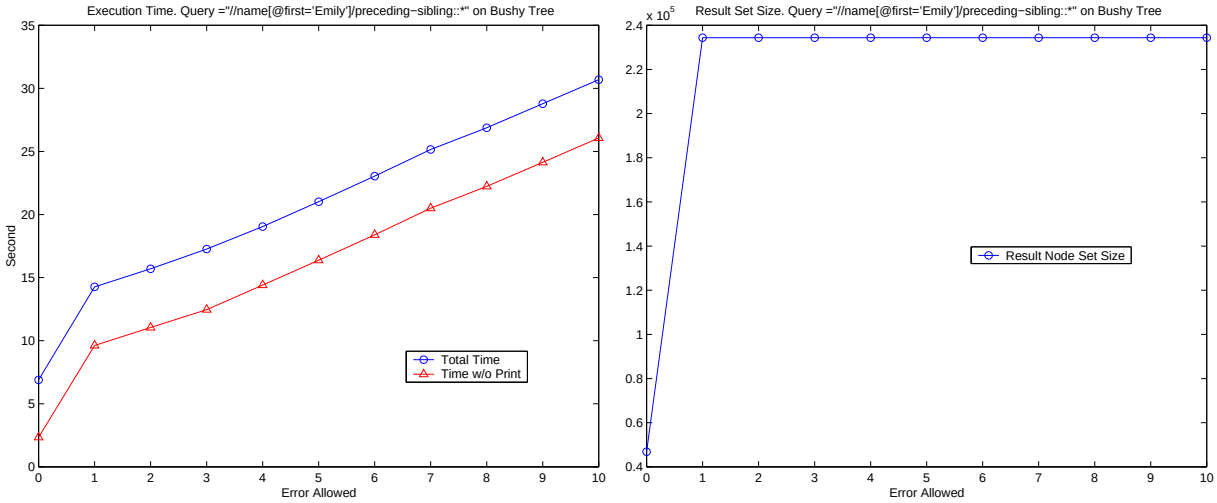

Figure 18: Testing query "//name[@first='Emily']/preceding-sibling::*" on bushy trees

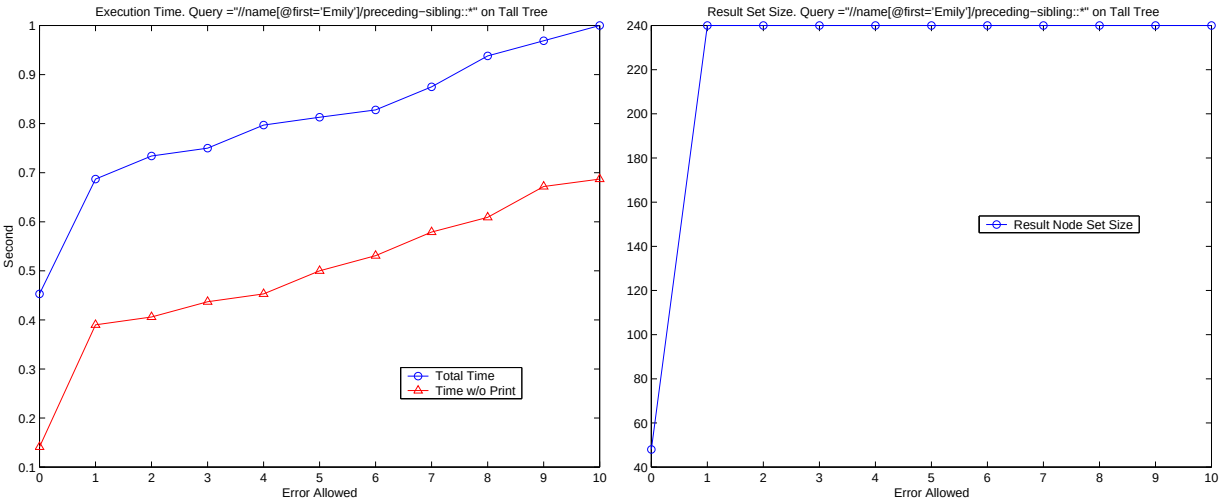

Figure 19: Testing "//name[@first='Emily']/preceding-sibling::*" on tall trees

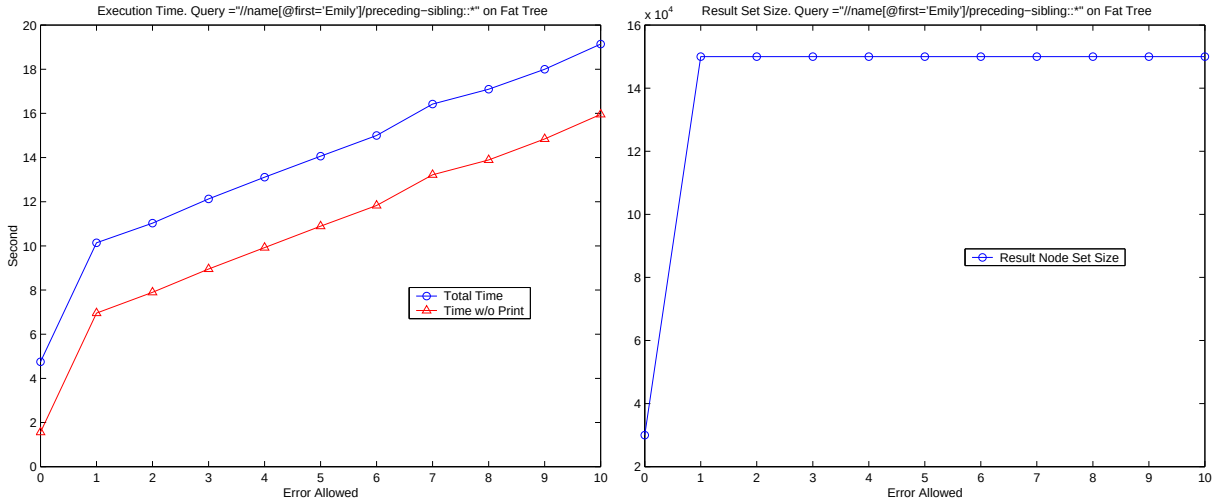

Figure 20: Testing query "//name[@first='Emily']/preceding-sibling::*" on fat trees

There is a proposal for a pattern matching language called approXQL (21; 23), which relaxes XQL (24) in three ways: node deletion, node insertion, and node renaming. By allowing only stylized sequences of deleting nodes (in a bottom-up fashion), they avoid the combinatorial effects of permitting arbitrary combinations of deletions. XIRQL (19) is an extension of XQL (24) which integrates IR features. XIRQL's features are weighting and ranking, relevance-oriented search (where only the requested content is specified and not the type of elements to be retrieved) and datatypes with vague predicates (e.g., search for measurements that were taken at about 30 feet). XXL (22) is a language inspired by XMLQL (25) that extends it for ranked retrieval. This extension consists of similarity conditions expressed using a binary operator that expresses the similarity between a value of a node of the XML data tree and a constant or an element variable given by a query. This operator can also be used for approximate matching of element and attribute names. Our work differs in that we focus on XPath. We think XPath is a better starting point because of its wide use, especially in XQuery. We also consider both hierarchical structure and content approximation and our results can be grouped according to edit distance metrics.

Two other recent papers focus on query relaxation in XPath (XQuery) (26; 1). Amer-Yahia et. al propose an efficient tree pattern relaxation technique based on three kinds of errors: node relabeling, edge generalization (converting a descendant to a child), and subtree promotion (taking a descendant subtree and moving it to a sibling. We've proposed a slightly different edit scheme for ApproXPath involving rewriting the axes. Amer-Yahia et. al develop a tree-pattern relaxation, that simultaneously considers all the location steps in an XPath expression (which is preferable for database applications). In contrast, we augmented a step-by-step, navigation-based approach (which is far less preferable for databases). We prefer their approach. Dongwon Lee's dissertation generalizes the tree pattern approach and provides an excellent overview of the entire field of query relaxation techniques for XML.

XML documents can be queried in a mediated environment by relaxing queries whose result is initially empty (27) Three kinds of relaxations are proposed: unfolding a node (replicating a node by creating a separate path to one of its children), deleting a node and propagating a condition at a node to its parent node. Unfortunately, this work does not consider any weighting and does not discuss evaluation techniques for relaxed queries.

Kanza and Sagiv (28) propose two semantics, flexible and semiflexible, for evaluating graph queries against a simplified version of the Object Exchange Model (OEM). Intuitively, under these semantics, query paths are mapped to database paths, so long as the database path includes all the labels of the query path; the inclusion need not be contiguous or in the same order; this is quite different from our notion of tree pattern relaxation. They identify cases where query evaluation is polynomial in the size of the query, the database and the result (i.e., combined complexity). However, they do not consider scoring and ranking of query answers.

In IR, there are three ways of controlling the set of relaxations that are applied to a query: threshold, top-k and boolean (c.f., (20; 22)) approaches. Most often, query terms are assigned weights based on some variant of the tf*idf method (12) and probability independence between elementary conditions is assumed. ApproXPath does not use any post-pruning method to limit the result set of a relaxation. All the irrelevant results are eliminated as soon as possible.

There exist two kinds of algorithms for approximate matching in the literature: post-pruning and rewriting-based algorithms. The complexity of post-pruning strategies depends on the size of query answers and a lot of effort can be spent in evaluating the total set of query answers even if only a small portion of it is relevant. Rewriting-based approaches can generate a very large number of rewritten queries. For example, the rewritten query might be quadratic in the size of the original query (21). Zhang's MPMGJN (29) algorithm is also similar to our approach. But we take different approach to the error measurement and provide more precise measurements on "similarity." She assigned different weight on the different nodes in the query tree and allowed limited relaxation on the query tree. Our approach utilizes tree distance metrics and relatively rich transformation on the tree.

Another related work is Approximate DataGuides (30), which extends DataGuides (31). Their approach is based on the similarity among sets and differs from our tree-metrics approach. Their focus is on extracting schemas for semistructured data, which differs from our focus on querying data.

7 Conclusions

In this paper, we propose ApproXPath, an approximate evaluation semantics for XPath. ApproXPath is intended for situations where users have some idea of the structure and content of an XML data collection, but may lack precise knowledge. ApproXPath relaxes the evaluation of axes, node tests, and predicates in XPath, allowing for a user-specified limit on the number of errors allowed. Each error corresponds to a tree edit operation. So by allowing one error, the user is searching not only the original XML tree, but also trees one edit operation away from the original. Query results are ranked by the number of errors, which gives users some feedback on how far from the original tree the node was located: nodes farther away likely have less relevance. ApproXPath does not change the syntax of XPath, and includes (as the no error case) its conventional semantics. Hence it can be used in place of any conventional XPath expression.

References

[1] D. Lee, *Query Relaxation for XML Model*. PhD thesis, Department of Comp. Science, UCLA, 2002.

[2] S. Wu and U. Manber, "Text searching allowing errors," *CACM*, vol. 35, pp. 83–91, Oct. 1992.

[3] S. Wu and U. Manber, "Fast text searching with errors," Tech. Rep. TR91-11, Department of Computer Science, University of Arizona, 1991.

[4] K.-C. Tai, "The tree-to-tree correcting problem," *JACM*, vol. 26, July 1979.

[5] D. T. Barnard, G. Clarke, and N. Duncan, "Tree-to-tree correction for document trees," Tech. Rep. 95-372, Department of Computing and Information Science, Queen's University, Jan. 95.

[6] K. Zhang, D. Shasha, and J. T. L. Wang, "Approximate tree matching in the presence of variable length don't cases," *Journal of Algorithms*, vol. 16, pp. 33 – 66, January 1994.

[7] H. Jin, *A framework for capturing, querying, and restructuring metadata in XML data*. PhD thesis, School of Electrical Engineering and

Computer Science, Washington State University, 2005.

[8] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of ACM*, vol. 21, pp. 168–173, Jan 1974.

[9] R. Lowrance and R. A. Wangner, "An extension of the string-to-string correction problem," *Journal of ACM*, vol. 22, pp. 177–183, April 1975.

[10] G. Navarro, "A guided tour to approximate string matching," *ACM Computing Surveys*, vol. 33, no. 1, pp. 31–88, 2001.

[11] P. Kilpelainen, *Tree Matching Problems with Application to Structured Text Database*. PhD thesis, Department of Computer Science, University of Helsinki, November 1992.

[12] G.Salton and M.J.McGill, *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, 1983.

[13] E. W. Brown, "Fast evaluation of structured queries for information retrieval," in *Proc. ACM SIGIR Conf.*, 1995.

[14] C. Faloutsos, "Access methods for text," *ACM Computing Surveys*, no. 1, 1985.

[15] M. Persin, "Document filtering for fast ranking," in *Proc. ACM SIGIR Conf*, (Dublin, Ireland), 1994.

[16] H. Turtle and J. Flood, "Query evaluation: Strategies and optimization," *Information Processing and Management*, pp. 831–850, Nov. 1995.

[17] W. Y. P. Wong and D. L. Lee, "Implementation of partial document ranking using inverted files," *Information Processing and Management*, no. 5, 1993.

[18] J. Zobel, A. Moffat, and R. SacksDavis, "An efficient indexing technique for fulltext database systems," in *VLDB*, 1992.

[19] N. Fuhr and K. Großjohann, "XIRQL: An extension of XQL for information retrieval," in *In ACM SIGIR Workshop On XML and Information Retrieval, Athens, Greece*, (http://citeseer.nj.nec.com/fuhr00xirql.html), July 2000.

[20] Y. Hayashi, J. Tomita, and G. Kikui, "Searching text-rich XML documents with relevance ranking," in *ACM SIGIR 2000 Workshop on XML and Information Retrieval*, (Athens, Greece), July 2000.

[21] T. Schlieder, "Similarity search in XML data using cost-based query transformations," in *ACM SIGMOD 2001 Web and Databases Workshop*, (Santa Barbara, California), May 2001.

[22] A. Theobald and G. Weikum, "Adding relevance to XML," *LNCS*, vol. 1997, pp. 105–131, 2001.

[23] T. Schlieder, "Approxql: Design and implementation of an approximate pattern matching language for XML," Tech. Rep. Report B 01-02, Institut fur Informatik, Freie Universitt Berlin, 2001.

[24] J. Robie, J. Lapp, and D. Schach, "XML query language (XQL)," in *Proceedings of QL'98 – The Query Languages Workshop*, (Cambridge, Mass.), 1998.

[25] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu, "A query language for XML," *Computer Networks (Amsterdam, Netherlands: 1999)*, vol. 31, no. 11–16, pp. 1155–1169, 1999.

[26] S. Amer-Yahia, S. Cho, and D. Srivastava, "Tree pattern relaxation," in *EDBT*, pp. 496–513, 2002.

[27] C. Delobel and M. Rousset, "A uniform approach for querying large tree-structured data through a mediated schema," in *International Workshop on Foundations of Models for Information Integration(FMII)*, (Viterbo, Italy), 2001.

[28] Y. Kanza and Y. Sagiv, "Flexible queries over semistructured data," in *Proceedings of the ACM Symposium on Principles of Database Systems*, 2001.

[29] C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo, and G. M. Lohman, "On supporting containment queries in relational database management systems," in *SIGMOD Conference*, 2001.

[30] R. Goldman and J. Widom, "Approximate dataguides," 1999.

[31] R. Goldman and J. Widom, "Dataguides:enabling query formulation and optimization on semitructured database," in *VLDB*, (Athens, Greece), pp. 436–445, August 1997.

A Base-line Experiment Queries

The following queries were used to compare the performance of Xalan to ApproXPath on a bushy kind of tree.

Query 1: /doc/level1/level2/level3/level4
Query 2: /doc/level1[@pos='1']/level2[@pos='2']/level3[@pos='3']/level4[@pos='4']
Query 3: /doc
Query 4: /doc/level1
Query 5: //name[@first='Emily']/preceding-sibling::*
Query 6: /doc/level1/level2/level3/level4/level5/level6
Query 7: /doc/level1[@pos='1']/level2[@pos='2']/level3[@pos='3']/level4[@pos='4']/ level5[@pos='5']/level6[@pos='5']/name[@first='David'][@last='Marston']

On a tall kind of tree, the following queries were used.

Query 1: /doc/level1/level2/level3/level4
Query 2: /doc/level1[@pos='1']/level2[@pos='1']/level3[@pos='1']/level4[@pos='1']
Query 3: /doc
Query 4: /doc/level1
Query 5: //name[@first='Emily']/preceding-sibling::*
Query 6: /doc/level1/level2/level3/level4/level5/level6
Query 7: /doc/level1[@pos='1']/level2[@pos='1']/level3[@pos='1']/level4[@pos='1']/ level5[@pos='1']/level6[@pos='1']//name[@first='David'][@last='Marston']'

On a fat kind of tree, the following queries were used.

Query 1: /doc/level1/level2
Query 2: /doc/level1[@pos='1']/level2[@pos='1']
Query 3: /doc
Query 4: /doc/level1
Query 5: //name[@first='Emily']/preceding-sibling::*
Query 6: /doc/level1/level2/name
Query 7: /doc/level1[@pos='1']/level2[@pos='1']/name[@first='David'][@last='Marston']