Efficient Processing of Branch Queries for High-Performance XML Filtering

Ryan H. Choi Raymond K. Wong
The University of New South Wales, Sydney, NSW, Australia
National ICT Australia, Sydney, NSW, Australia
{ryanc,wong}@cse.unsw.edu.au

ABSTRACT

In this paper, we consider the problem of filtering a continuous stream of XML data efficiently against a large number of branch XPath queries. Several approaches have been proposed, and many of them improve their run-time efficiencies by sharing some paths between branch queries. This paper further improves the run-time efficiencies by classifying and grouping semantically equivalent twig patterns, and identifying the common paths that are shared between these groups. Query structure matching is done at index compilation phase, and the paths shared between these groups of queries are processed once. Experiments show that our proposal is efficient and scalable compared to previous work.

1. INTRODUCTION

XML streaming processing has become an important research field due to the increasing developments of a wide range of applications that require efficient evaluation of XML streaming data. These applications include automotive telematics, precision agriculture, defense systems, telemedicine and processing of scientific data [9, 17, 16]. One common functionality that these applications must provide is how to evaluate a large number of multiple complex user queries against a set of data coming from streaming continuously. A system which implements such a functionality is known as a publish-subscribe system (also denoted as a pub-sub system). With the popularity of XML as a data exchange format, there have been several research efforts that address the problems of building scalable and efficient XML filtering systems in which users express their interests of data in XPath.

In this paper, we consider pub-sub systems which already know the structure of XML documents (i.e., DTD) that they process, and propose techniques that utilize such information. Subscribers of these pub-sub systems already have such information and use the knowledge to formulate specific XPath queries for a particular set of incoming XML documents. One characteristic of such systems is, they process many but similar XML documents that share a common DTD against a set of XPath queries which are specifically designed for that group of XML documents. The probability of this set of queries being matched also tends to be higher than a set of general and unspecific queries that are formulated without such information. Many previous works ([4, 5, 6, 23, 18, 15, 3]) did not consider how queries with different probabilities of matching could affect the performances of pub-sub systems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
INFOSCALE 2007, June 6-8, Suzhou, China
Copyright © 2007 ICST 978-1-59593-757-5
DOI 10.4108/infoscale.2007.213

Using the information on document structures, we build a query index for a specific set of documents that share the same DTD, and pre-process given queries against the query index prior to process incoming XML documents. An advantage of this approach is, most of query matching process can be done at index compilation phase, and therefore only the minimum operations required to evaluate queries are performed in run time.

Unlike some previous works, we do not split queries with branch paths—which we name them as *branch queries*—into multiple smaller linear queries. In our approach, branch queries are treated as twig patterns and evaluated to true when there are documents that match the same twig patterns. This approach, however, requires us to solve several challenging problems in order to make it more efficient than previous approaches. For instance, consider the following queries, `q1:a/b[d]/c/e` and `q2:a/b[c/e]/d`, where the twig patterns of the two queries are equivalent, but expressed in different forms. Classifying a set of queries into smaller sets of queries such that each set contains queries whose twig patterns are the same is important. This is because the effect of evaluating one query from a set is the same as evaluating all queries in the same set. For example, if `q1` and `q2` were in the same set, and `q1` was evaluated to true, all other queries including `q2` would also be evaluated to true without processing each of them. In this way, the scalability of the system is not proportional to the total number of queries but proportional to the *unique* number of twig patterns that represent queries. In addition, the process of identifying queries must be more efficient than the one which evaluates queries individually without performing such optimization.

Another challenging problem is that, there exist queries with different twig patterns that look for the same parts of a document. For example, a query `q3:a/b[f]/c/e` is different from `q1` (and `q2`), but also looks for the same branch, `a/b/c/e`. Identifying such branch sharing and eliminating any duplicate evaluation of branch paths can improve the

overall performance of a system, since it reduces the total number of branches that have to be evaluated in run time. For example, the effect of processing the branch `/a/b/c/e` once should be the same as evaluating the branch in `q1`, `q2` and `q3` individually. Similarly, the cost of the branch identifying process should be less than the cost of processing similar queries individually.

In this paper, we propose efficient and scalable XML filtering techniques which also address the above problems. Our techniques are different from previous approaches in a way that we pre-process branch queries at index compilation phase to reduce duplicate processing of queries and share partial results amongst queries when they are evaluated. At index compilation phase, branch queries are applied to our query index as twig patterns, and in run time, only containment relationships between elements are checked using our label scheme. Before processing branch queries, we classify queries into groups, each of which has the same twig patterns. Moreover the common paths that are shared by these groups of queries are identified, and in run time, only the unique twig patterns and branch paths are evaluated. By utilizing the information about document structures, both run time and the scalability of the system are increased compared to previous work. In addition, unlike approaches such as [23, 18], we do not consider predicate ordering as we believe there are more applications that require to ignore such ordering. For example, we treat queries `a/b[c][d]` and `a/b[d][c]` the same.

This paper is organized as follows. Section 2 describes related work. Section 3 describes a data structure and a SAX parser upon which our technique is built. Section 4 describes our techniques for processing branch queries. Section 5 presents the experimental results of our techniques. Finally, we conclude our paper in Section 6.

2. RELATED WORK

XFilter [1] is one of the earliest pub-sub systems in which each XPath expression is modeled as a finite state machine. However, it does not support branch XPath expressions, and is designed towards handling small discrete XML documents which makes it less effective against continuous stream of XML documents.

XTrie [4, 5] is based on a trie which supports branch XPath queries. It decomposes twig patterns into longest substrings and uses a trie to detect occurrences of substring matches for each event it receives. Decomposing to longest substrings also lowers the probability of matching documents. Constructing an index of queries is similar to our approach, but in our approach, query index is built on top of a document structure. During index compilation phase, shared paths between queries are also identified. In addition, our index provides faster element matching at run time as there are fewer number of nodes to look up.

YFilter [7, 6] is a successor of XFilter which supports branch XPath queries. It improves the performance by sharing some paths between branch queries, and also separates filtering problem of XPath expressions into structural matching and content matching parts. To achieve path sharing, a Nondeterministic Finite Automaton of branch queries is built. Similar to XTrie, branch queries are decomposed into multiple linear queries, and in the post-processing phase, the results from the structure matching part are combined. In this approach, the processing cost is dominated in the post-processing phase when all results are combined. Branch queries are evaluated individually in the post-processing phase whereas our approach only evaluates queries that have different twig patterns.

LazyDFA [13] shows how it evaluates a large number of linear XPath queries using Deterministic Finite Automaton. Its experiments demonstrate that it takes a constant processing time independent of the query size. One drawback of this approach is the space requirement which is not bounded as pointed out in their experiments. However, the size of LazyDFA is kept minimal by building DFA at run time lazily which results in having small document structures for data-oriented documents. Its approach is efficient both in time and space for evaluating linear XPath expressions for data-oriented documents. Our approach can also be applied to process linear queries at constant time.

XPush [14] extends LazyDFA to process branch XPath queries using deterministic pushdown automaton in a bottom up way. Although, in theory, it takes constant time to process each incoming element, it rarely achieves its theoretical performance due to a large requirement of memory. In practice, the efficient of this approach is about linear to the number of queries.

The work by Onizuka [20] is an improvement of LazyDFA which reduces the number of states required to process complex XML documents by grouping linear queries into several clusters, and supports branch queries by building NFA which is shared by DFA states. Similar to other approaches, it also decompose branch queries and evaluate each linear query by LazyDFA. As a result, the performance depends on the number of event invocations from its XPath processor, which also depends on the number of decomposed XPath expressions. However, it shows how effective a DFA approach can be compared to others.

Unlike previous approaches, Bloom Filter [12] improves the efficiency of filtering XML data by sacrificing the accuracy of returning matching queries for incoming documents. It uses Bloom filters to represent a group of linear queries, and provides a hash-based approach to evaluate XPath expressions. Queries with the same prefix are shared to decrease the number of candidates. However, it does not support branch queries and is not suitable for applications that allow any inaccuracy.

XSQ [21, 22] also builds a NFA for a branch query and maintains its own buffer to store potential matching elements. This allows to return a set of matching elements for a given query. However, only one query can be processed at a time for an incoming document, and the buffer can be overflowed for queries that return a large number of elements such as the entire document.

PRIX [23] and its successor FiST [18] consider branch XPath expressions and XML documents as twig patterns, and transform each branch query into a sequence of elements using a tree preorder traversal or Prufer Sequence repectively. It then matches the sequence of incoming elements against sequences of branch queries. It does not require to decompose branch queries into linear queries and therefore combining them at a post-processing phase. It organizes the sequences into a hash based index to improve efficiency. However, the implicit ordering created using the preorder traversal or Prufer Sequence algorithm imposes the implicit predicate ordering for branch queries. For example, a query `a/b[c]` is matched only if element `b` appears before

c under a.

XPath-NFA [15] is a simplified version of YFilter which builds NFA from XPath queries, but it converts NFA to DFA through subset construction, and organizes automata in a way that it utilizes CPU's L2 cache. It identifies the area where state transition occurs the most frequently in an automaton, and places these area into an in-memory data structure, called hot buffer, for fast cache access. Hot buffers can also be configured to support incremental updates of XPath queries.

AFilter [3] builds a reverse directed graph, prefix tree and suffix tree from XPath queries, and when queries are evaluated, query results are shared amongst queries that have the same prefix. By looking at the suffix tree, it reduces the number of graph traversals needed when queries with the same prefix are evaluated. However, it does not support branch XPath queries.

There are some works that use XML algebra to efficiently process queries. The work by Fegaras [10] transforms a query into XML algebras to optimize it according to a query plan, and processes XML documents that are fragmented by its Hole-Filter approach. Raindrop [25] also transforms a query into XML algebras, but uses NFAs to encode path expressions in a query. Both approaches are for XQueries, and cannot process multiple queries at a time.

Unlike all pub-sub systems above, which are light-weight in-memory based, some systems such as [24, 30, 27] use relational database to support XML document filtering. In these systems, XML subscriptions and documents are stored in relational databases as tuples, and join operations are performed to find matching subscriptions for each document. The advantage of this approach is, the scalability of the systems is not limited to the amount of physical memory that systems have, and since all subscriptions are stored in databases, the systems are not volatile. However, the performance of this approach is also substantially slower compared to other in-memory approaches.

In general, the problem of evaluating XPath expressions efficiently can be solved by taking DFA approaches such as LazyDFA and using path sharing concepts such as YFilter. In addition, the performance can be further improved by grouping queries that represent the same twig patterns as it provides a way of eliminating duplicate processing of queries and therefore reducing the total number of queries to process. However, none of the previous approaches identifies queries that have the same twig patterns, and evaluate only unique twig patterns using the paths that are shared between these twig patterns.

3. PRELIMINARY

In this section, we introduce a data structure called Structure Index and a modified SAX parser upon which our technique is built. Let us first define terminologies that we use in the following sections. A *query tree* is a tree representation of a branch query. We name each location step of a query a *node*. A query tree contains two types of nodes—(1) main nodes which are the nodes that represent the main path of a query, and (2) *predicate* nodes that represent the branch paths of a query. A node is a *leaf* node if it does not have any child nodes. A node is a *branch* node if it has at least one predicate node as a child.

```
<a m="1">
  <f>
    <g>2</g>
    <g>3</g>
  </f>
  <e>
    <c>
      <d>4</d>
      <f>5</f>
      <d>6</d>
    </c>
    <a>text</a>
  </e>
  <f/>
  <b>
    <f>7</f>
    <e>8</e>
  </b>
</a>
```

(a) XML document

```
<a>
  <@m/>
  <f>
    <g/>
  </f>
  <e>
    <c>
      <d/>
      <f/>
    </c>
    <a/>
  </e>
  <b>
    <f/>
    <e/>
  </b>
```

(b) Structure Index

```
startElem("a")
  startElem("@m")
  endElem("@m")
  startElem("f")
    startElem("g")
    endElem("g")
    startElem("g")
    endElem("g")
  endElem("f")
  startElem("e")
    startElem("c")
      startElem("d")
...
endElem("a")
```

(c) Modified SAX events

Figure 1: Sample XML document, Structure Index and modified SAX events

3.1 Structure Index

Structure Index is a minimum document structure representation of a set of XML documents that share the same DTD. It contains only unique element and attribute names, and ignores (1) document ordering, and (2) both content of elements and values of attributes. Figure 1(a) and 1(b) show an example of an XML document and its Structure Index. It is different from DTD in a way that DTD provides the complete information about a document structure of an XML document whereas Structure Index simplifies DTD by removing recursive elements by expanding these elements up to the maximum depth of incoming documents. It is similar to DataGuides [11] and ViST [29], but Structure Index is used to extract data structures of documents in order to preprocess and expand queries whereas the other indexes are used to index data to improve query processing. The size and complexity of Structure Index is small and simple in practice for data-oriented documents [11], even though its DTD allows more complicated structures such as infinitely recursive elements. A study in [19] also discovered that, 99% of a sample of 200,000 documents publicly available on the Web have less than 8 levels of nesting. At index compilation phase, each node in the Structure Index, denoted as *Index Node*, is decorated with information that is needed to process branch queries at runtime.

Before processing documents, the Structure Index is generated from a set of training documents, all of which represent various document structures of incoming documents to filter. They can be randomly sampled from publishers or generated randomly using a tool such as [8]. We refer to the initial period of time during which the Structure Index is constructed as *update phase*. We also refer to the stage where updates of Structure Index do not or hardly occur as *stable phase*.

3.2 SAX Parser and Element Processing

A SAX parser is modified in the following ways. `startElem(elem)` and `endElem(elem)` process attributes as they were child elements of the current element. Figure 1(c) shows the events produced when a document in Figure 1(a)

is parsed.

Initially, a current Index Node is set to the root of the Structure Index. On `startElem(elem)` event, an Index Node whose name is the same as *elem* is searched amongst the children of the current Index Node. Only when such a child node is found, that child node becomes the current Index Node. The searching process takes $O(1)$ since the names of child nodes are hashed.

On `endElem(elem)` event, the parent of the current Index Node becomes the current Index Node if the name of the *elem* is the same as the current Index Node and *elem* occurs at the same level as the current Index Node. Each time when an event is received, branch queries associated with the current Index Node are processed. We describe our query processing techniques in the following sections.

4. METHODS

In this section, we present a technique which is optimized for evaluating a set of branch queries against a sequence of incoming XML documents. The technique can also be simplified to efficiently evalute linear queries. Before we process branch queries, we label Index Nodes in the Structure Index using a label scheme such as [2], and decorate the nodes with the information that is obtained from the following three steps. In addition, we keep a global stack of elements whose end tags have not yet received. An element is pushed or popped from the stack when a `startElem(elem)` or `endElem(elem)` is received respectively. Each stack item stores a set of labels that we discuss shortly.

4.1 Identifying Equivalent Twig Patterns

Figure 3 shows an example of a Structure Index from Figure 1(b) with branch queries from Figure 2 applied to it. The attribute node representing `@m` is omitted in the figure for simplicity. There are two cases in which a leaf or branch node of a query tree is assigned to more than one Index Node. First, it is when a query contains either '`//`'s between nodes or '`*`'s in a query, and second, it is when Structure Index has more than one set of matching nodes of the query. We use the term, *query instance*, to refer to one set of matching Index Nodes of a query tree. In other words, a query tree can have more than one query instance if it satisfies one of the two conditions above. Conversely, each Index Node contains a set of matching leaf nodes or branch nodes of unique query instances. Linear queries only contain leaf nodes and they are applied similarly.

We identify groups of query instances, each of which contains the equivalent twig patterns as follows. For each query tree, we identify groups of query instances, each of which contains a set of matching leaf and branch nodes. For each query instance, we sort the Index Nodes in the query instance according to their Structure Index labels, and append all decomposed paths of the query tree. We then insert (k, v) pair to a multi-hashtable, where k is a sorted sequence of labels of Index Nodes plus decomposed paths, and v is a query ID. We repeat the process for all queries.

The resulting multi-hashtable, which is shown in Table 1, contains groups of queries, and the queries in each group represent the same twig pattern for this set of documents. This is because they require the same set of leaf and branch Index Nodes to be satisfied for this particular set of documents, and require the same set of nodes to be structured in the same way. The table contains two rows of `q7`, since `q7` has two query instances. One may say that identifying query instances could increase the number of queries to process, but by grouping these query instances, we reduce the total processing time, since the number of groups of query instances is smaller than the total number of queries, and evaluating one query from each group is sufficient to evaluate all other queries in the same group. Furthermore, we do not evaluate query instances once they have been reported as matched. Algorithm 1 outlines the above procedures.

Query ID	Branch Query
1	/a[f][b/f]/e/c[f]/d
2	/a[f][b/e]/e/c[f]/d
3	/a[f/g][e]/e[a]//d
4	//a[b//f][f]//c[d]/f
5	//a[f][.//e]//c[f]/d
6	/a[.//g][.//e]/e[a]//d
7	/a[e]//f

Figure 2: Branch queries

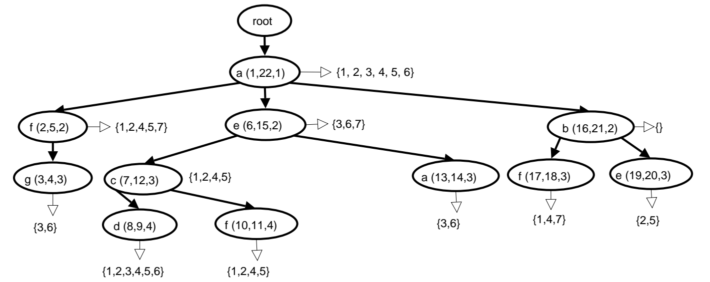


Figure 3: Identifying semantically equivalent branch queries

Table 1: An example of a multi-hashtable

Label	Query ID
(1,22,1)(2,5,2)(7,12,3)(8,9,4)(10,11,4)(17,18,3) (/a/e/c/d,/a/b/f,/a/e/c/f,/a/f)	1,4
(1,22,1)(2,5,2)(7,12,3)(8,9,4)(10,11,4)(19,20,3) (/a/e/c/d,/a/b/f,/a/e/c/f,/a/f)	2,5
(1,22,1)(3,4,3)(6,15,2)(8,9,4)(13,14,3) (/a/e/c/d,/a/e,/a/e/a,/a/f/g)	3,6
(1,22,1)(2,5,2)(6,15,2)(/a/f,/a/e)	7
(1,22,1)(6,15,2)(17,18,3)(/a/b/f,/a/e)	7

4.2 Identifying Common Paths Between Queries

Although all queries in the Structure Index represent unique twig patterns, we observe that some paths between leaf and branch nodes, or branch and branch nodes are shared by many other twig patterns. For example, in Figure 3, both `q1` and `q2` require to have all `c`, `d` and `f` nodes to be matched even though these queries represent different twig patterns.

Algorithm 1 $ClassifyTwigs(SIRootNode, queryTrees)$

1: $multihashtable \leftarrow \{\}$, $twigs \leftarrow \{\}$
2: **for** $q_i \in queryTrees$ **do**
3: $\quad \{QueryInstance_{1..n}\}$
$\quad \leftarrow applyQuery(SIRootNode, q_i)$
4: $\quad$ **for** $qi_i \in \{QueryInstance_{1..n}\}$ **do**
5: $\quad\quad sort(qi_i)$
6: $\quad\quad \{paths_{1..n}\} \leftarrow getDecomposedQueries(qi_i)$
7: $\quad\quad key_i \leftarrow getLabel(qi_i) \cup \{paths_{1..n}\}$
8: $\quad\quad multihashtable.add(key_i, qi_i)$
9: **for** $key_i \in multihashtable$ **do**
10: $\quad \{q_{1..n}\} \leftarrow multihashtable.get(key_i)$
11: $\quad twigs.add(combineQueryTrees(\{q_{1..n}\}))$
12: **return** $twigs$

In such a case, we group these queries, and when either nodes `c` and `d`, or `c` and `f` are visited, we notify that either the path between `c` and `d`, or `c` and `f` respectively are matched for both queries `q1` and `q2`. Note that `q3` cannot be in the same group as `q1` and `q2` as it requires a different branch node (i.e., node `e`). The process of grouping is applied recursively from leaf nodes towards the root node of the Structure Index.

The effect of this grouping is that, we instantly identify which paths of what queries are matched by processing that path only once. Let us name such a group a *path group*. Figure 4 illustrates such a grouping in which *gID*:$\{i..j\}$=>$\{n..m\}$ notation is used, where *gID* is a path group ID, $i..j$ are query IDs that are in the same group, and $n..m$ are the list of expecting labels which are explained in the next section. In the figure, at Index Nodes `c`, `d` and `f`, queries `q1` and `q2` are in the same path group. This allows us to immediately identify that some paths of `q1` and `q2` are matched when node `f` is visited. Algorithm 2 outlines the above procedures.

Algorithm 2 $IdentifyCommonPaths(node)$

1: //*node* is a Structure Index node
2: **if** $!node.isLeaf()$ **then**
3: $\quad$ **for** $child_i \in node.children()$ **do**
4: $\quad\quad IdentifyCommonPaths(child_i)$
5: $queryGroups \leftarrow \{\}$ // is of type multi-hashtable
6: **for** $queryTreeNode_i \in node.getQueryTreeNodes()$ **do**
7: $\quad branchNode$
$\quad \leftarrow getClosestBranchNode(queryTreeNode_i)$
8: $\quad$ **if** $branchNode! = null$ **then**
9: $\quad\quad queryGroups.insert(branchNode, queryTreeNode_i)$
10: $\quad$ **else**
11: $\quad\quad node.addPathGroup($
$\quad\quad createPathGroup(queryTreeNode_i))$
12: **for** $val_i \in queryGroups$ **do**
13: $\quad node.addPathGroup(createPathGroup(val_i))$

4.3 Building Sets of Expecting Labels

Having identified path groups of all queries, we construct a set of expecting labels for each path group in an Index Node. Each path group contains Index Node labels that satisfy the following—(1) the Index Node labels are from the Index Nodes that match either leaf or branch nodes of all query trees inside the path group, and (2) these Index Nodes are the closest descendant Index Nodes from the current Index Node, where the path group is in. Figure 4 illustrates such groups of labels inside path groups located in Index Nodes. An expecting label is in *INLabel:pID* format, where *INLabel* is a Index Node label, and *pID* is an ID of a path group.

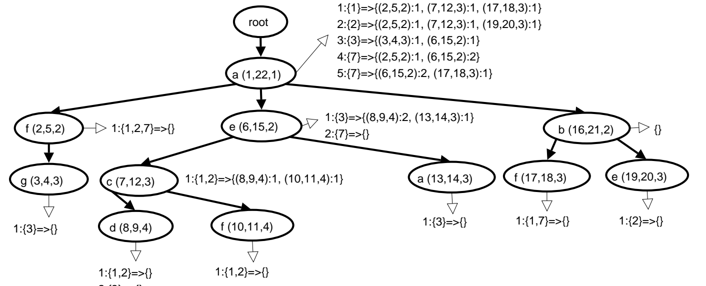


Figure 4: A Structure Index for branch queries

In this figure, the path group with pID 1 inside Index Node `(1,22,1)` requires three expecting labels, `(2,5,2):1`, `(7,12,3):1` and `(17,18,3):1`. That means, to match `q1`, all three path groups with pID 1 in `(2,5,2)`, `(7,12,3)` and `(17,18,3)` Index Nodes must be matched. A path group in an Index Node is matched in two ways. Firstly, if the path group does not contain any expecting labels, it is matched when the Index Node is visited. Secondly, if the path group contains expecting labels, it is matched only if all of its expecting labels have been matched. Finally, expecting labels for each path group is recursively constructed. In addition, leaf nodes do not have any expecting labels as they do not have any descendant nodes.

4.4 Processing Queries

After building the Structure Index through the above three steps, the matching process of groups of queries now becomes a simple checking process of labels against the set of expecting labels for each path group. On `startElem(`*`elem`*`)` event, we create a set of *INLabel:pID* labels for each path group. A stack item representing the current element is pushed to a global stack, and the set of these labels are inserted to each stack item of the global stack. Linear queries are reported as matched if leaf nodes of linear queries are found in the current Index Node.

On `endElem(`*`elem`*`)` event, we iterate the path groups in the current Index Node, and for each path group, we check whether all of its expecting labels can be found from the top element of the stack. For the case where all expecting labels of a path group are found, we report all queries in that path group as being matched. The process is repeated for each path group. Finally, the top stack item is popped. By popping the top stack item, all labels that are no longer valid are removed. Note that we only process a path between nodes once and identify all queries that share the same path. Algorithm 3 and 4 outline the above procedures.

Algorithm 3 $startElem(elem)$

1: $labels \leftarrow \{\}$
2: $currentNode \leftarrow currentNode.getChild(elem.getName())$
3: $stack.push(currentNode)$
4: **for** $pathGroup_i \in node.getPathGroups()$ **do**
5: $\quad labels.add(makeLabel(node, pathGroup_i))$
6: **for** $stackItem_j \in stack$ **do**
7: $\quad stackItem.add(labels)$

Algorithm 4 $endElem(elem)$

1: $matchedQueries \leftarrow \{\}$
2: **for** $pathGroup_i \in currentNode.getPathGroups()$ **do**
3: $match \leftarrow true$
4: **for** $eLabel_j \in pathGroup_i.getExpectingLabels()$ **do**
5: **if** $!stack.top().found(eLabel_j)$ **then**
6: $match \leftarrow false$
7: $break$
8: **if** $match = true$ **then**
9: $matchingQueries.add(pathGroup_i.getQueries())$
10: $stack.pop()$
11: $currentNode \leftarrow currentNode.parent()$
12: **return** $matchedQueries$

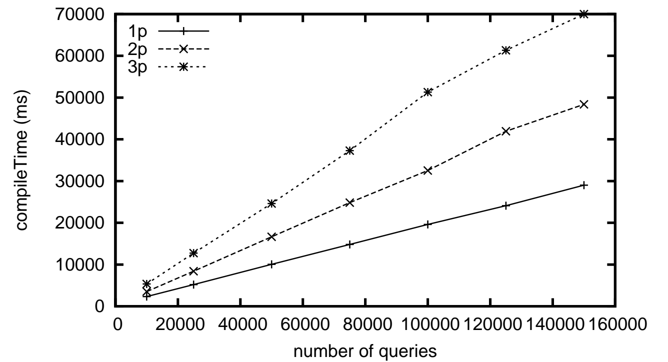


Figure 5: Compilation time vs. number of queries for various number of branches per query

5. EXPERIMENTS

We now present experimental results to support our approaches. All experiments were executed on a Pentium 4 3.2GHz machine with 1Gb ram running Ubuntu Linux. Our approaches were implemented in Java 1.5. The SAX parser we used was Xerces Java Parser 2.8.0 [26].

5.1 Settings

5.1.1 Documents

We used NASA dataset obtained from [28], and adopted the experimental settings used for documents from [18]. In this setting, each dataset was split into smaller documents, and categorized into three datasets, each of which had ranges [10kb,20kb), [20kb,30kb) and [30kb,60kb) in size. We then randomly selected 250 documents from each dataset.

5.1.2 Queries

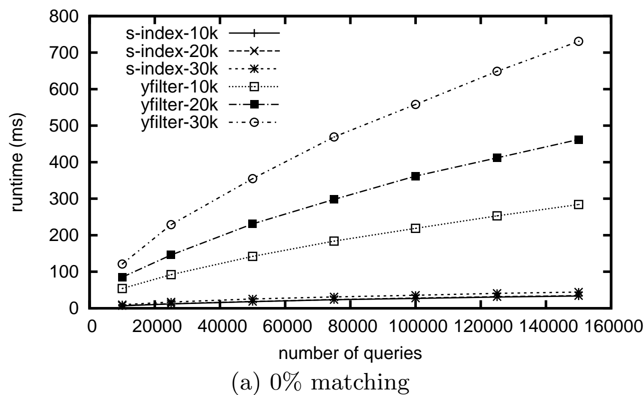
We generated all queries for the experiments in the following ways. Firstly, the set of XML documents obtained from Section 5.1.1 was parsed and a super set of document structures that represented all documents were extracted. Secondly, a random element from the document structure was selected, and all nodes on the path between that selected node and the root node were scanned. While scanning nodes on the path, we picked a node with a probability of 80% as well as maintaining the ordering to the root node. We also picked a node with a probability of 10% and replaced that node with a '*'. For each selected node, we also assigned '`//`' with a probability of 10%. From those selected elements, we randomly picked p number of nodes which were to contain branch paths. The branch path was generated in the similar way except the node which would contain branch path was treated as a root node. Lastly, all duplicate queries were removed, and the second step was repeated until q number of queries were produced.

In order to control the percentage of matching queries, after generating q number of queries, we obtained a set of queries that had at least one matching document by filtering the documents generated from Section 5.1.1 against these randomly generated queries. We also obtained a set of queries that did not match any documents in the similar way. These sets of queries were mixed to create queries with various matching probabilities. This approach gives many advantages over previous ways of generating queries such as in [6]. By looking at the actual document structure rather than entirely rely on the DTD of a document, it only generates queries with the elements that actually occur, and the depth of each query is no more than the maximum depth of matching documents even though a DTD of a document allows infinite depth. More importantly, it allows us to control the probability of matching queries for a set of documents.

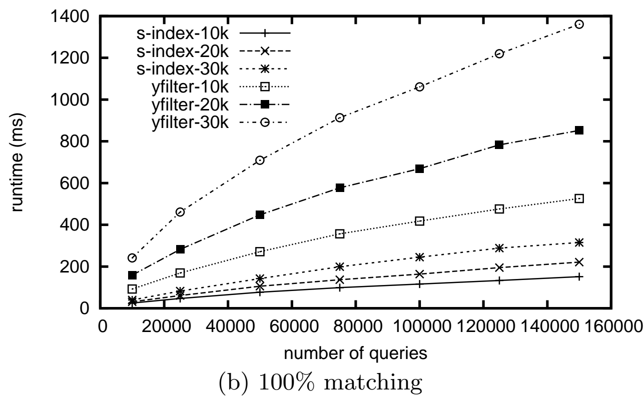
5.2 Results

Figure 5 shows the experimental results about the time required to create Structure Index for various number of queries with various number of predicates at index compilation phase. The compilation time increases as the number of queries increases since the operations required to identify groups of queries is proportional to the number of queries. Similarly, as the number of branch paths for each query increases, the compilation time also increases. This is due to the increased number of labels and path groups that are required to process branch queries. For the following experiments, the Structure Index created in this phase is used, and all reported times are the processing times measured while queries are evaluated in stable phase.

Figure 6 shows the run times to evaluate various number of queries for various sizes of datasets. The probabilities of queries that match a set of documents is set to 0% and 100% respectively and each query has 2 branches. The run times reported here are the average times taken to process 250 random documents. In addition, we have also measured the run time performance of YFilter [6] (excluding the time required for its index compilation) and compared it against our system. In this experiment, both systems show different run times for queries with 0% and 100% matching probabilities. Both systems perform better when the probability of matching documents is low. This is because, for our system, the number of label checking for `startElem()` and `endElem()` are reduced, and for YFilter, the number of state transition for `startElem()` and `endElem()` and operations required for the post-filtering are reduced. When the matching probability is low, our system processes a large number of branch queries very efficiently. This is because there are a small number of labels that are successfully stored on top of a stack. This makes the set operation very efficient. Moreover, since a query is only satisfied when all its branch paths are satisfied, a short cut evaluation can easily be applied. In a short cut evaluation, a query is failed and is not further processed once a failure of any of its branch paths is detected. Similar patterns of graphs are observed when queries with



(a) 0% matching



(b) 100% matching

Figure 6: Runtime vs. number of queries for various sizes of data

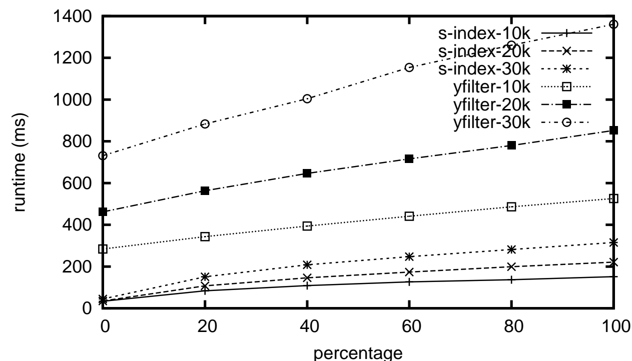


Figure 7: Runtime vs. various probability of matching queries for various sizes of data

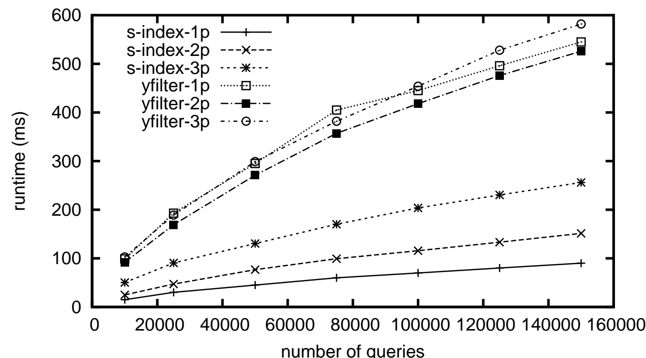


Figure 8: Runtime vs. number of queries for various number of branches per query

partial matches (i.e., 20%, 40%, etc...) are evaluated by both systems.

Figure 7 shows the run time to process various probability of 150,000 queries with 2 branches against each group of datasets. Our system shows a rapid increase for queries with between 0% and 20% matching probabilities. This is because a fair number of labels required to process queries are generated in that range. The rate of increase in run time is slowed down from around 40% as most labels required to process queries have already been generated and many number of labels have started being shared between queries. As the number of labels that are shared increases, the number of newly generated labels decreases. Our system also shows a constant run time increase for datasets that only differ in size. Compared to YFilter, the constant run time increase is smaller.

Figure 8 shows the run time to process various number of queries against different number of branches. In this experiment, we used 10k dataset, and the probability of matching document was set to 100%. The run time increase of our system for queries with different number of branches is due to the increased number of label checking at run time. The graph also shows that by removing duplicate evaluation of both queries and paths, the rate of increase in run time of our system is lower as the number of queries increases.

6. CONCLUSION

We have presented an efficient approach of evaluating a large number of branch XPath queries on XML streaming data. Our approach uses Structure Index to classify branch queries into groups of individual twig patterns, and identifies the common paths that are shared between these groups. These common paths are then processed by using our labels to efficiently evaluate groups of queries that have the same common paths. Our experiments show that our approach can evaluate a large number of branch queries, and is more scalable and efficient than the previous research work such as YFilter.

7. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, San Francisco, CA, September 2000. Morgan Kaufmann.

[2] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal xml pattern matching. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 310–321, Madison, WI, June 2002. ACM.

[3] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura,

and D. Agrawal. Afilter: Adaptable xml filtering with prefix-caching and suffix-clustering. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 559–570, Seoul, Korea, September 2006. ACM.

[4] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. In *Proceedings of the 18th International Conference on Data Engineering*, pages 235–244, San Jose, CA, February 2002. IEEE Computer Society.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of xml documents with xpath expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[7] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. Yfilter: Efficient and scalable filtering of xml documents. In *Proceedings of the 18th International Conference on Data Engineering*, pages 341–342, San Jose, CA, February 2002. IEEE Computer Society.

[8] A. L. Diaz and D. Lovell. Xml generator, September 1999. http://www.alphaworks.ibm.com/tech/xmlgenerator.

[9] D. Estrin and et al. *Embedded everywhere: A research agenda for networked systems of embedded computers.* Computer Science and Telecommunications Board. National Academy Press, 2001.

[10] L. Fegaras, D. Levine, S. Bose, and V. Chaluvadi. Query processing of streamed xml data. In *Proceedings of the 11th International Conference on Information and Knowledge Management*, pages 126–133, McLean, VA, November 2002. ACM.

[11] R. Goldman and J. Widom. Dataguides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of 23rd International Conference on Very Large Data Bases*, pages 436–445, Athens, Greece, August 1997. Morgan Kaufmann.

[12] X. Gong, Y. Yan, W. Qian, and A. Zhou. Bloom filter-based xml packets filtering for millions of path queries. In *Proceedings of the 21st International Conference on Data Engineering*, pages 890–901, Tokyo, Japan, April 2005. IEEE Computer Society.

[13] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing xml streams with deterministic automata and stream indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[14] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 419–430, San Diego, CA, June 2003. ACM.

[15] B. He, Q. Luo, and B. Choi. Cache-conscious automata for xml filtering. *IEEE Transactions on Knowledge and Data Engineering*, 18(12):1629–1644, 2006.

[16] F. Hu, Y. Wang, and H. Wu. Mobile telemedicine sensor networks with low-energy data query and network lifetime considerations. *IEEE Transactions on Mobile Computing*, 5(4):404–417, April 2006.

[17] Q. Huang, S. Bhattacharya, C. Lu, and G.-C. Roman. Far: Face-aware routing for mobicast in large-scale sensor networks. *ACM Trans. Sen. Netw.*, 1(2):240–271, 2005.

[18] J. Kwon, P. Rao, B. Moon, and S. Lee. Fist: Scalable xml document filtering by sequencing twig patterns. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 217–228, Trondheim, Norway, August 2005. ACM.

[19] L. Mignet, D. Barbosa, and P. Veltri. The xml web: a first study. In *Proceedings of the 12th International World Wide Web Conference*, pages 500–510, Budapest, Hungary, May 2003. ACM.

[20] M. Onizuka. Light-weight xpath processing of xml stream with deterministic automata. In *Proceedings of the 12th International Conference on Information and Knowledge Management*, pages 342–349, New Orleans, LA, November 2003. ACM.

[21] F. Peng and S. S. Chawathe. Xpath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 431–442, San Diego, CA, June 2003. ACM.

[22] F. Peng and S. S. Chawathe. Xsq: A streaming xpath engine. *ACM Trans. Database Syst.*, 30(2):577–623, 2005.

[23] P. Rao and B. Moon. Prix: Indexing and querying xml using prüfer sequences. In *Proceedings of the 20th International Conference on Data Engineering*, pages 288–300, Boston, MA, March 2004. IEEE Computer Society.

[24] P. Seshadri. Building notification services with microsoft sqlserver. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 635–636, San Diego, CA, June 2003. ACM.

[25] H. Su, J. Jian, and E. A. Rundensteiner. Raindrop: a uniform and layered algebraic framework for xqueries on xml streams. In *Proceedings of the 12th International Conference on Information and Knowledge Management*, pages 279–286, New Orleans, LA, November 2003. ACM Press.

[26] The Apache XML Project. Xerces2 java parser, last accessed: 2006. http://xerces.apache.org/xerces2-j/.

[27] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable xml publish/subscribe system using a relational database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 479–490, Paris, France, June 2004. ACM.

[28] University of Washington. Xml data repository, last accessed: 2006. http://www.cs.washington.edu/research/xmldatasets/.

[29] H. Wang, S. Park, W. Fan, and P. S. Yu. Vist: A dynamic index method for querying xml data by tree structures. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 110–121, San Diego, CA, June 2003. ACM.

[30] H. Zeller. Nonstop sql/mx publish/subscribe: Continuous data streams in transaction processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, page 636, San Diego, CA, June 2003. ACM.