Efficient Filtering of XML Documents with XPath Expressions

Chee-Yong Chan, Pascal Felber, Minos Garofalakis, Rajeev Rastogi
Bell Laboratories, Lucent Technologies
{cychan,pascal,minos,rastogi}@research.bell-labs.com

Abstract

We propose a novel index structure, termed XTrie, that supports the efficient filtering of XML documents based on XPath expressions. Our XTrie index structure offers several novel features that make it especially attractive for large-scale publish/subscribe systems. First, XTrie is designed to support effective filtering based on complex XPath expressions (as opposed to simple, single-path specifications). Second, our XTrie structure and algorithms are designed to support both ordered and unordered matching of XML data. Third, by indexing on sequences of element names organized in a trie structure and using a sophisticated matching algorithm, XTrie is able to both reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering. Our experimental results over a wide range of XML document and XPath expression workloads demonstrate that our XTrie index structure outperforms earlier approaches by wide margins.

1. Introduction

The exploding volume of information (e.g., stock quotes, news reports, advertisements) made available on the Internet has fueled the development of a new generation of applications based on *selective data dissemination*, where specific data is selectively relayed to a large number (e.g., millions) of distributed clients. This trend has led to the emergence of novel middleware architectures that asynchronously propagate data from a set of *publishers* (i.e., data generators) to a large number of widely dispersed *subscribers* (i.e., data consumers) who have pre-registered their interest in specific information items [4]. In general, such *publish-subscribe* architectures are implemented using a set of networked servers that selectively propagate relevant messages to the consumer population, where message relevance is determined by *subscriptions* representing the consumers' interests in specific messages.

The majority of existing publish/subscribe systems have typically relied on simple subscription mechanisms, such as keyword or "bag of words" matching, or simple comparison predicates on attribute values. For example, systems such as Gryphon [1], Siena [4], and Elvin [15], all use filters in the form of a set of attributes and simple arithmetic or boolean comparisons on the values of these attributes. The recent emergence of XML (eXtensible Markup Language) [18] as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription/filtering mechanisms that exploit both the *structure* and the *content* of published XML documents. In particular, the XPath language [17], which is a W3C proposed standard for addressing parts of an XML document, has been adopted as a filter-specification language by a number of recent XML data dissemination systems (e.g., XFilter [2], Intel's NetStructure XML Accelerator [6]). Given the increased complexity of structural, XPath-based data filters, the problem of effectively identifying the subscriptions that match an incoming XML document poses a difficult and important research challenge. More specifically, the key problem faced in XPath-based data-dissemination systems can be abstracted as the following *XPath Expression (XPE) Retrieval Problem*: "Given a large collection $\mathcal{P}$ of XPath expressions (XPEs) and an input XML document D, find the subset of XPEs in $\mathcal{P}$ that match D."

The key technique for expediting XPE retrieval is to construct an appropriate index structure on the given collection of XPE subscriptions. Since XPEs can, in general, represent structurally complex tree patterns, building index structures for efficient XPE retrieval is a non-trivial problem. Furthermore, simplistic approaches (e.g., building an index based solely on the element names contained in the XPEs) can result in very ineffective retrieval schemes that incur a lot of unnecessary checking of (irrelevant) XPE subscriptions.

In this paper, we propose a novel index structure, termed *XTrie*, that supports the efficient filtering of XML documents based on XPath expressions. Our XTrie index structure offers several novel features that make it especially attractive for large-scale publish/subscribe systems. First, XTrie is designed to support effective filtering based on complex XPath expressions (as opposed to simple, single-path specifications). Second, our XTrie structure and algorithms are designed to support both ordered and unordered matching of XML data. Note that ordered matching is an important requirement for many applications (e.g., document processing) that has typically been overlooked in existing data dissemination systems. Third, by indexing on sequences of element names (i.e., *substrings*) organized in a *trie structure* and using a sophisticated matching algorithm, XTrie is able to both reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering.

Indexing on a carefully-selected set of substrings (rather than individual element names) in the XPEs is a key ingredient of our approach that enables us to minimize both the number and the cost of the required index probes. The key intuition here is that a sequence of element names has a lower probability (compared to a single element name) of matching in an input document, resulting in fewer index probes. In addition, since there are fewer indexed XPEs associated with a "longer" substring key, each index probe is likely to be less expensive as well.

To support on-line filtering of streaming XML data, our XTrie indexing scheme is based on the event-based SAX parsing interface [13], to implement XML data filtering as the XML document is parsed. This is in contrast to the alternative DOM parsing interface [16], which requires a main-memory representation of the XML data tree to be built before filtering can commence. To the best of our knowledge, the only other SAX-based index structure for the XPE retrieval problem is Altinel and Franklin's XFilter [2], which relies on indexing the XPE element names using a hash-table structure. By indexing on substrings rather than individual element names, our XTrie index provides a much more effective indexing mechanism than XFilter. A further limitation of XFilter is that its space requirement can grow to a very large size as an input document is parsed, which can also increase the filtering time significantly. Our experimental results over a wide range of XML document and XPath expression workloads validate our claims, demonstrating that our XTrie index structure significantly outperforms XFilter (by factors of up to 4).

2. Background

XPath Expressions (XPEs) and XPE-trees. An XML document comprises a hierarchically nested structure of *elements*, starting with a root element; sub-elements of an element can themselves be elements and can also contain character data (i.e., text) and attributes. Elements can be nested to any depth and the scope of an element in the XML document is defined by a start-tag and an end-tag. The XPath language treats XML documents as a tree of nodes (corresponding to elements) and offers an expressive way to specify and select parts of this tree. XPath expressions (XPEs) are structural patterns that can be matched to nodes in the XML data tree. The evaluation of an XPE yields an object whose type can be a node-set, a boolean, a number, or a string. For our XPE retrieval problem, an XML document matches an XPE when the evaluation result is a non-empty node set.

The simplest form of XPEs specify a single-path pattern, which can be either an absolute path from the root of the document or a relative path from some known location (i.e., context node). A path pattern is a sequence of one or more *location steps*. In its basic form, a location step specifies a node name (i.e., an element name), and the hierarchical relationships between the nodes are specified using parent-child ("$/$") operators (i.e., at adjacent levels) and ancestor-descendant ("$//$") operators (i.e., separated by any number of levels). For example, the XPE $/a/b//c$ selects all c element descendants of all b elements that are direct children of the root element a in the document. XPath also allows the use of a wildcard operator ("$*$") to match any element name at a location step.

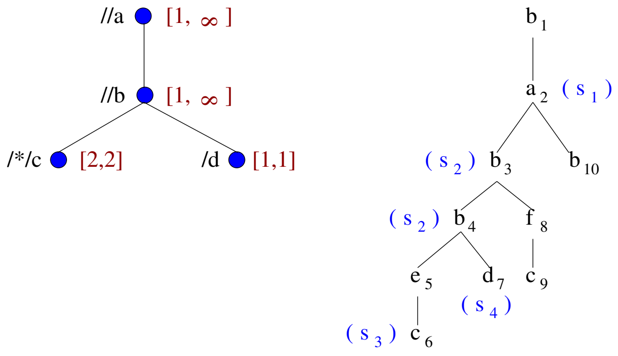
Each location step can also include one or more predicates to further refine the selected set of nodes. Predicate expressions are enclosed by "[" and "]" symbols. The predicates can be applied to the text or the attributes of the addressed elements, and may also include other path expressions. Any relative paths in a predicate expression are evaluated in the context of the element nodes addressed in the location step at which they appear. For example, the XPE $/a[b[@x \geq 100]/c]/*/d$ specifies a tree pattern starting at the root element a with two child "branches" b/c and $*/d$ such that the element b has an attribute x with a value equal to or greater than 100.

The tree pattern specified by an XPE can be represented by an ordered rooted tree, where each node is labeled with an element name (prefixed by either "$/$" or "$//$" followed by an optional sequence of one or more "*/"). The ordering of the child nodes for each parent node is based on their order of appearance in the XPE. We refer to such a tree representation of an XPE as an *XPE-tree*.

Unordered and Ordered XPE Matchings. Before we describe the two modes of matching XPEs, we first introduce some new definitions and notation. Given two nodes v and v' in a rooted tree T, we say that v *precedes* v' in a post-order traversal of T, denoted by $v \prec_{post} v'$, if v is visited before v' in a post-order traversal of T.

Each node d in an XML document tree is associated with a *level number*, denoted by $level(d)$, where $level(d) = 1$ if d is the root element; otherwise, $level(d) = level(d') + 1$, where d' is the parent node of d.

Each node t in an XPE-tree T is associated with a *relative level* (with respect to its parent node in T), which is defined to be at least k, denoted by $relLevel(t) = [k, \infty]$, if the label of t is prefixed with "$//$" followed by $(k-1)$ "$*$"; otherwise, if the label of t is prefixed with "$/$" followed by $(k-1)$ "$*$", then the relative level of t is defined to be



(a) **(b)**

Figure 1. Unordered and Ordered Matchings.

exactly k, denoted by $relLevel(t) = [k, k]$.

Consider an XPE-tree T with the set of nodes $\{t_1, t_2, \cdots, t_m\}$ and an XML document tree D. We say that a node t_i in T *matches* at a node d in D if the element name of t_i is equal to that of d. In the *unordered matching model*, where T is treated as an unordered tree, T *matches* D if there exists a set of m nodes $\{d_1, d_2, \cdots, d_m\}$ in D such that (1) for each node t_i in T, t_i matches at d_i, and (2) for each child node t_j of a node t_i in T, d_j is a descendant of d_i such that $level(d_j) - level(d_i) \in relLevel(t_j)$. As an example, consider the XPE-tree T of $p = //a//b[*/c]/d$ in Figure 1(a), where the label and relative level of each node are indicated on its left and right, respectively; and the XML document tree D in Figure 1(b), where the subscripts indicate the order in which the nodes are parsed (ignore the parenthetical annotations for now). Note that T matches D with $//a$, $//b$, $/*/c$, and d matching at a_2, b_4, c_6, and d_7, respectively.

In addition to the model of unordered matchings, XPath also allows the order of matching to be explicitly specified. Consider again the XPE-tree in Figure 1(a) for p. If we wish to indicate that the "branch" $*/c$ must match in the document before the "branch" d, this can be expressed using the XPE $p' = //a//b/*[following\text{-}sibling::d]/c$. Referring again to Figure 1, if the positions of the two subtrees rooted at e_5 and d_7 in D are swapped, then p' would not match D while p would still match D. In the *ordered matching model*, where T is treated as an ordered tree, T *matches* D if (1) T matches D in the unordered matching model, and (2) for each pair of child nodes t_j and t_k of each internal node in T, $t_j \prec_{post} t_k$ in T iff $d_j \prec_{post} d_k$ in D.

Note that *hybrid matchings* of XPEs, which involve both unordered as well as ordered matchings, are also possible. Due to space constraints, we shall focus on only ordered matchings of XPEs that do not contain any attributes in the rest of this paper. Details on handling attributes as well as unordered and hybrid matchings are given in [5].

3. XPE Decompositions and Matchings

In this section, we describe the mechanisms employed in our XTrie index for decomposing XPEs into sequences of XML element names (i.e., *substrings*), and define several important concepts for matching based on substring-trees that play a key role in our XTrie indexing structure and matching algorithms.

3.1. Substring Decompositions

Given an XPE p, we define a sequence of element names $s = t_1.t_2.\cdots.t_n$ to be a *substring* of p if s is equal to the concatenation of the element names of the nodes along a path $< v_1, v_2, \cdots v_n >$ in the XPE-tree of p, such that each v_i is the parent node of v_{i+1} ($1 \leq i < n$) and the label of each v_i (except perhaps for v_1) is prefixed only by "/". In other words, each pair of consecutive element names in a substring of p must be separated by a parent-child ("/") operator. We use $Path(s)$ to denote the path of nodes in the XPE-tree of p that defines the substring s. As an example, consider the XPE $p = /a/b[c/d//e][g//e/f]//*/*/e/f$ whose XPE-tree is depicted in Figure 2(a). The set of substrings of p includes abg, bcd, ef and b; on the other hand $abge$, gef, and bef are not substrings of p, since they involve an intermediate element name (i.e., e) that is not prefixed by "/".

A sequence of substrings $S = < s_1, s_2, \cdots, s_n >$ of an XPE p is said to be a *substring decomposition of* p if each $s_i \in S$ is a substring of p and each node t_j in p's XPE-tree is contained in $Path(s_i)$ for some $s_i \in S$. The ordering of the substrings in S is fixed based on the order in which they would be matched in an ordered matching of p; i.e., s_i should be matched before s_{i+1}. A substring decomposition S is a *minimal decomposition of* p if each substring s_i of S is of maximal length; that is, there does not exist another longer substring in p's XPE-tree that contains s_i. Clearly, a minimal decomposition of p comprises the smallest possible number of substrings among all possible decompositions of p. Figures 2(a) and (b) show two possible substring decompositions for our example XPE p, where each dashed region encloses a path of nodes defining a substring. Note that S_a is the (unique) minimal decomposition of p.

Our XTrie index relies on substring decompositions for installing XPEs into the indexing structure. The choice of a specific class of substring decompositions impacts both the space and performance of the index. Minimal decompositions, in particular, have two important performance advantages. First, since longer substrings have a lower probability of being matched in the input XML document, the maximal-length substrings chosen in a minimal decomposition generally result in fewer index probes. Second, since there are fewer XPEs associated with a longer substring, the cost of each index probe is generally lower with mini-

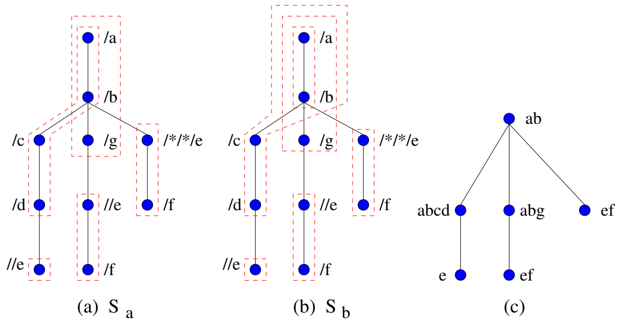


Figure 2. Substring Decompositions.

mal decompositions. On the other hand, using only a minimal decomposition for an XPE can result in problems when checking for an unordered matching. For example, consider again the minimal decomposition S_a in Figure 2(a), where $s_1 = abcd$, $s_2 = e$, $s_3 = abg$, $s_4 = ef$, and $s_5 = ef$. Since "ab" is part of s_1 and s_3 but not part of s_5, for unordered matching, using only S_a would fail to detect a matching of p when s_5 matches after "ab" has been matched but before s_1 and s_3 are matched.

Intuitively, to avoid such problems, we need to enrich the minimal decomposition of an XPE so that it "takes note" of the branching nodes in the XPE-tree. Our XTrie indexing scheme accomplishes this through the use of *simple XPE decompositions*. Formally, a substring decomposition S is said to be a simple decomposition of an XPE p if S can be partitioned into two sequences S_1 and S_2, where: (1) S_1 is the minimal decomposition of p; and, (2) S_2 consists of one substring s for each *branching node* v in p's XPE-tree, such that s is the maximal substring in p with v as its last node and s is not already listed in S_1. As an example, the decomposition S_b depicted in Figure 2(b) is the simple decomposition of our example XPE p; note that S_b simply adds the substring ab (b is a branching node) to the minimal decomposition S_a. Also, note that, for a single-path XPE, its simple decomposition is equal to its minimal decomposition.

The substrings of the simple decomposition of p_i can be organized into a unique rooted tree, referred as the *substring-tree* of p_i, as follows. Let $S_i = < s_{i,1}, s_{i,2}, \cdots, s_{i,|p_i|} >$ denote the simple decomposition of p_i, where $|p_i|$ denotes the number of substrings in the simple decomposition of p_i. Then, the *root substring* is $s_{i,1}$, and the *parent substring* of $s_{i,j}$, where $j > 1$, is $s_{i,k}$ (or equivalently, $s_{i,j}$ is the *child substring* of $s_{i,k}$) if either (1) $Path(s_{i,k})$ is a prefix of $Path(s_{i,j})$, or (2) the last node of $Path(s_{i,k})$ is the parent node of the first node of $Path(s_{i,j})$ in the XPE-tree of p_i. The ordering among sibling substrings is based on their ordering in S_i. As an example, Figure 2(c) shows the substring-tree for the simple decomposition in Figure 2(b). A substring that has no child substrings is called a *leaf substring*. A substring $s_{i,j}$ is said to be a *descendant* of another substring $s_{i,k}$ if either $s_{i,k}$ is the parent substring of $s_{i,j}$, or the parent substring of $s_{i,j}$ is a descendant of $s_{i,k}$. Similarly, $s_{i,k}$ is said to be an *ancestor* of $s_{i,j}$ if $s_{i,j}$ is a descendant of $s_{i,k}$. Finally, we define the *rank of a substring* $s_{i,j}$ to be equal to k if $s_{i,j}$ is the k^{th} child of its parent substring; the rank of the root substring is 1.

We now extend the notion of relative level that was defined for nodes in XPE-trees to substrings. Informally, the relative level of a substring s refers to the relative difference in levels between the last elements of s and its parent substring in a matching. More formally, consider a substring s of an XPE p (with parent substring s'), and let $t = < t_1, t_2, \cdots, t_n >$ be the longest suffix of $Path(s)$ such that $t_1 \notin Path(s')$. Let $relLevel(t_i) = [\ell_i, u_i]$ for $1 \leq i \leq n$, and let k denote $\sum_{i=1}^{n} \ell_i$. Then the *relative level* of s is defined to be at least k, denoted by $relLevel(s) = [k, \infty]$, if $\max_{1 \leq i \leq n}\{u_i\} = \infty$; otherwise, it is defined to be exactly k, denoted by $relLevel(s) = [k, k]$.

3.2. Matching with Substrings

Consider an XML document tree D, and an XPE p_i with XPE-tree T_i and simple decomposition $< s_{i,1}, s_{i,2}, \cdots, s_{i,|p_i|} >$. Since each substring $s_{i,j}$ corresponds to some path of nodes $Path(s_{i,j})$ in T_i, we can extend the definition of matching for nodes to substrings as follows: $s_{i,j}$ *matches* at a node d in D (or there is a *matching* of $s_{i,j}$ at d in D) if $Path(s_{i,j})$ matches D such that the last node of $Path(s_{i,j})$ matches at d. We say that there is a matching of $s_{i,j}$ at level ℓ in D if $s_{i,j}$ matches at some node at level ℓ in D.

As the nodes in D are parsed in a pre-order traversal (by the SAX parser), the ordered matching of p_i in D also progresses incrementally following a pre-order traversal of the substring-tree of p_i such that each substring $s_{i,j}$ is matched before $s_{i,k}$, $k > j$. Thus, to determine if p_i matches D, we need to keep track of the "partial matchings" of p_i in D. However, since we are interested only in whether or not p_i matches D and not in the actual number of match occurrences, "partial matchings" of p_i that are "redundant" should be ignored in order to improve the effectiveness of the filtering process.

We now formally define the notions of partial and redundant matchings. Given an XPE p_i and an XML document tree D, we define M_i to be a *set of matchings* (with respect to p_i and D) if M_i contains pairs of the form $(s_{i,j}, d_j)$, where $s_{i,j}$ matches at d_j, and for each distinct pairs $(s_{i,j}, d_j)$, $(s_{i,j'}, d_{j'}) \in M_i$, $s_{i,j} \neq s_{i,j'}$ and $d_j \neq d_{j'}$. We say that there is a *partial matching* of $s_{i,j}$ at a node d_j in D if there exists a set of matchings M_i such that for each $1 \leq k \leq j$, (1) $(s_{i,k}, d_k) \in M_i$; and (2) for each child substring $s_{i,k'}$ of $s_{i,k}$, $d_{k'}$ is a descendant of d_k such that $level(d_{k'}) - level(d_k) \in relLevel(s_{i,k'})$. It follows that we have a *(complete) matching* of p_i in D if there is a partial matching of $s_{i,|p_i|}$ at some node in D. We represent a partial matching by its set of matchings M_i.

To define redundant matching, we first need to introduce the notion of subtree-matching. A set of matchings M_i is said to be a *subtree-matching* of $s_{i,j}$ if M_i is a partial matching of each descendant of $s_{i,j}$. Informally, a partial matching of $s_{i,j}$ at a node d is considered redundant if there exists a subtree-matching of $s_{i,j}$ at some "earlier" node d' (i.e., $d' \prec_{post} d$ in D); thus, all subsequent partial matchings that require the matching of $s_{i,j}$ at d can be safely ignored without affecting the correctness of deciding whether or not p_i matches D. More precisely, a partial matching of $s_{i,j}$ at d_j (represented by M_i) is said to be a *redundant matching* if there exists a subtree-matching of $s_{i,k}$ (represented by $M_i{}'$), where $s_{i,k}$ is either $s_{i,j}$ itself or an ancestor of $s_{i,j}$, such that (1) $(s_{i,j}, d_{j'}) \in M_i{}'$ and $d_{j'} \prec_{post} d_j$ in D; and (2) if $s_{i,k}$ is not the root substring of p_i, then $(s_{i,k'}, d_{k'}) \in M_i \cap M_i{}'$, where $s_{i,k'}$ is the parent substring of $s_{i,k}$. Otherwise, M_i is said to be a *non-redundant matching* of $s_{i,j}$.

Consider again the XPE p and XML document D in Figure 1, where the four substrings in the simple decomposition of p are: $s_1 = a$, $s_2 = b$, $s_3 = c$, and $s_4 = bd$. The parenthetical annotation "(s_j)" besides a node d_i in D means that there is a non-redundant matching of s_j at d_i when d_i is parsed in D. Thus p matches D. Both the partial matchings of s_3 at c_9 and s_2 at b_{10} are redundant. Observe that a non-redundant matching could later become redundant as more nodes in the document tree are parsed; in particular, the non-redundant matching of s_2 at b_3 becomes redundant after d_7 is parsed.

4. The XTrie Indexing Scheme

In this section, we present our novel XTrie indexing scheme for filtering XML documents based on XPEs. Due to space constraints, we focus only on ordered matchings. The details for unordered and hybrid matchings can be found in [5].

4.1. The Index Structure

Let $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$ denote the set of XPEs being indexed, and $\mathcal{S}$ denote the set of distinct substrings derived from all the simple decompositions of the XPEs in $\mathcal{P}$. An XTrie index consists of two key components: (1) a *Trie* [12] (denoted by T) constructed on $\mathcal{S}$ to facilitate detection of substring matchings in the input XML data; and, (2) a *Substring-Table* (denoted by ST) that stores information about each substring of each XPE in $\mathcal{P}$. The information in ST is used to check for partial matchings. We now describe each of these two XTrie components in detail.

The Substring-Table. The substring-table ST contains one row for each substring of each indexed XPE; i.e., there are $\sum_{p \in \mathcal{P}} |p|$ rows in ST with each row corresponding to some $s_{i,j}$. The rows in ST are physically clustered in terms of the XPEs such that the substrings belonging to an XPE p are stored in consecutive rows ordered based on the simple decomposition of p. The order of the XPEs in ST is arbitrary. Since each row r in ST corresponds to some substring, for convenience, we use the notation $r_{i,j}$ to the denote the row in ST that corresponds to the substring $s_{i,j}$.

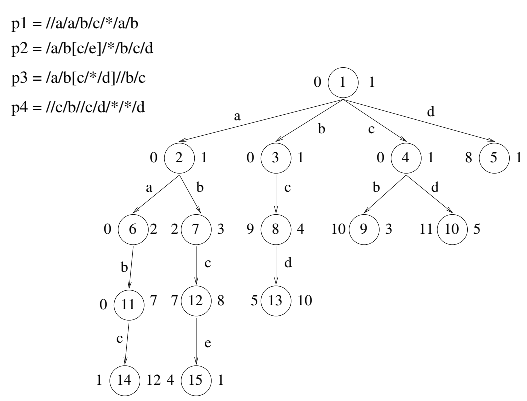
To facilitate locating all XPEs that contain some substring, the rows in ST are also logically partitioned into $|\mathcal{S}|$ disjoint blocks such that each block contains all the rows that correspond to the same substring. This substring-based partitioning of the rows in ST is achieved by chaining the rows within each block using a singly linked list, giving a total of $|\mathcal{S}|$ singly linked lists in ST (with one list for each distinct substring in $\mathcal{S}$). The rows within each linked list are partially ordered such that if rows $r_{i,j}$ and $r_{i,k}$ belong to the same linked list, then $r_{i,k}$ precedes $r_{i,j}$ in the linked list if $j < k$. This is required to ensure correctness under the ordered matching model [5].

Each row in ST (corresponding to some substring $s_{i,j}$) is a 5-tuple $(ParentRow, RelLevel, Rank, NumChild, Next)$, where:

- $ParentRow$ refers to the row number of the tuple in ST corresponding to the parent substring of $s_{i,j}$. ($ParentRow = 0$ if $s_{i,j}$ is a root substring.)
- $RelLevel$ is the relative level of $s_{i,j}$ (i.e., $relLevel(s_{i,j})$).
- $Rank$ is the rank of $s_{i,j}$ (i.e., $Rank = k$ if $s_{i,j}$ is the k^{th} child substring of its parent substring).
- $NumChild$ is the total number of child substrings of $s_{i,j}$.
- $Next$, which is a "pointer" for a singly linked list, is the row number of the next tuple in ST that belongs to the same logical block as the current row. If the current row is the last row in the linked list, then $Next = 0$.

The Trie. The trie T is a rooted tree constructed from the set of distinct substrings $\mathcal{S}$, where each edge in T is labeled with some element name. Each node N in T is associated with a label, denoted by $label(N)$, which is the string formed by concatenating the edge labels along the path from the root node of T to node N; the label of the root node is an empty string. T is constructed such that for each $s \in \mathcal{S}$, there is a unique node N in T such that $label(N) = s$; and for each leaf node N in T, $label(N) \in \mathcal{S}$. In addition to the pointers to nodes at the next level of the trie, each node N in T has two special pointers:

- The *Substring pointer* (denoted by $\alpha(N)$) points to some row in ST (i.e., $\alpha(N)$ is a row number) as follows: if $label(N) \in \mathcal{S}$, then $\alpha(N)$ points to the first row of the linked list associated with substring $label(N)$; otherwise, $\alpha(N) = 0$.



	Parent Row	Rel Level	Rank	Num Child	Next
1	0	$[4,\infty]$	1	1	0
2	1	$[3,3]$	1	0	3
3	0	$[2,2]$	1	2	6
4	3	$[2,2]$	1	0	0
5	3	$[4,4]$	2	0	0
6	0	$[2,2]$	1	2	0
7	6	$[1,1]$	1	1	0
8	7	$[2,2]$	1	0	12
9	6	$[2,\infty]$	2	0	0
10	0	$[2,\infty]$	1	1	0
11	10	$[2,\infty]$	1	1	0
12	11	$[3,3]$	1	0	0

Figure 3. XTrie Example.

- The *Max-suffix pointer* (denoted by $\beta(N)$) points to some internal node in T and its purpose is to ensure the correctness of the matching algorithm. Specifically, $\beta(N) = N'$ if $label(N')$ is the longest proper suffix of $label(N)$ among all the internal nodes in T; if N' does not exist, then $\beta(N)$ points to the root node of T.

Figure 3 depicts the XTrie index structures for a set of four XPEs $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$, where their respective simple decompositions are as follows: $S_1 = < aabc, ab >$, $S_2 = < ab, abce, bcd >$, $S_3 = < ab, abc, d, bc >$, and $S_4 = < cb, cd, d >$. The number within each trie node N represents the node's identifier; and the values of $\alpha(N)$ and $\beta(N)$ are shown to the left and right of N, respectively.

4.2. The XTrie Matching Algorithm

Our XTrie indexing scheme is designed to support online filtering of streaming XML data and is based on the SAX event-based interface that reports parsing events. Figure 4 shows the search procedure for the XTrie, which accepts as input an XML document D and an XTrie index (ST, T), processes the parsing events generated by D, and returns the identifiers of all the matching XPEs in the index.

The basic idea of the search algorithm is as follows. We use the trie T to detect the occurrence of matching substrings as the input document is parsed. For each matching substring s detected, we iterate through all the instances of s in the indexed XPEs (by traversing the appropriate linked list of rows in the substring-table ST associated with s) to check if the matched substring s corresponds to any non-redundant matching. Since the information stored in ST is static, we need to maintain some additional dynamic run-time information to ensure that we check only for non-redundant matchings.

This run-time information is maintained in the form of a two-dimensional integer-array $\mathcal{B}$ of size $|ST| \times L_{max}$, where $|ST|$ denotes the number of rows in the substring-table ST, and L_{max} is the maximum number of levels in an XML document. $\mathcal{B}[r_{i,j}, \ell] = n$, $n > 0$, if there is a non-redundant matching of $s_{i,j}$ (represented by M) at level ℓ such that the n^{th} child substring of $s_{i,j}$ is the leftmost child substring of $s_{i,j}$ for which a subtree-matching has not yet been detected (i.e., M is a subtree-matching of the $(n-1)^{th}$ child substring of $s_{i,j}$ if $n > 1$). Each $\mathcal{B}[r_{i,j}, \ell]$ is initialized to 0, and is incremented to 1 after a non-redundant matching of $s_{i,j}$ at level ℓ is detected. As more substring matchings are detected, the value of $\mathcal{B}[r_{i,j}, \ell]$ is incremented from n to $n+1$, $n > 1$, when the matching M also becomes a subtree-matching of the n^{th} child substring of $s_{i,j}$. The value of $\mathcal{B}[r_{i,j}, \ell]$ is reset to 0 when the end-tag corresponding to the begin-tag at level ℓ is parsed. Note that since $\mathcal{B}$ is a large sparse array, its implementation can be optimized to minimize space (e.g., using linked lists).

To understand how $\mathcal{B}$ is used to detect non-redundant matchings, suppose that a matching of substring $s_{i,j}$ at level ℓ has been detected, and $s_{i,j}$ is the n^{th} child substring of $s_{i,k}$. This matching is a partial matching of $s_{i,j}$ if there exists a matching of $s_{i,k}$ at level ℓ' such that $\ell - \ell' \in relLevel(s_{i,j})$ and $\mathcal{B}[r_{i,k}, \ell'] \geq n$. If, in addition, the value of $\mathcal{B}[r_{i,k}, \ell']$ is exactly n, then this partial matching is non-redundant; otherwise, it is redundant and it can safely be ignored. We know that an XPE p_i matches the input document when $\mathcal{B}[r_{i,1}, \ell] = m + 1$ for some value of ℓ, where m is the number of child substrings of the root substring $s_{i,1}$.

Our XTrie `SEARCH` algorithm (depicted in Figure 4) begins by initializing the search node N to be the root node of the trie T (Step 5). For each start-tag t encountered, if there is an edge out of N with the label t (to another trie node N' in T), the search continues on node N'. For each trie node N' visited, a matching substring (corresponding to $label(N')$) is detected if $\alpha(N') \neq 0$; in this case, Algorithm `MATCH-SUBSTRING` is invoked to process the matching substring using the substring table ST. Furthermore, for each trie node N' visited, we also need to check for other potential matching substrings that are suffixes of $label(N')$; this is achieved by using the max-suffix pointer (i.e., $\beta(N')$) in Step 16. On the other hand, if there is no edge out of a node N with the current tag t, this means that the concatenation of $label(N)$ and t is not a matching substring. Therefore, we need to check for other potential matching

Algorithm SEARCH (D, ST, T)
Input: D is an input XML document. (ST, T) is an XTrie index.
Output: R is the set of XPEs that matches D.
1) Initialize R to be empty;
2) Initialize $Node[i]$ = root node of T for $i = 0$ to L_{max};
3) Let $\mathcal{B}$ be a $|ST| \times L_{max}$ integer-array with all values initialized to 0;
4) Initialize $\ell = 0$; // ℓ is the current document level
5) Initialize N to be the root node of T; // N is the current trie node
6) **repeat**
7) **if** (a start-tag t is parsed in D) **then**
8) $\ell = \ell + 1$;
9) **while** ((there is no edge labeled "t" from N) **and** (N is not the root node of T)) **do**
10) $N = \beta(N)$;
11) **if** (there is an edge labeled "t" from N to N' in T) **then**
12) $Node[\ell] = N = N'$;
13) **while** (N' is not the root node) **do**
14) **if** ($\alpha(N') > 0$) **then**
15) $R = R \cup$ MATCH−SUBSTRING($ST, \mathcal{B}, \alpha(N'), \ell$);
16) $N' = \beta(N')$;
17) **else if** (an end-tag is parsed in D) **then**
18) Reset $\mathcal{B}[i, \ell]$ to 0 for i = 1 to $|ST|$;
19) $Node[\ell]$ = root node of T;
20) $\ell = \ell - 1$;
21) $N = Node[\ell]$;
22) **until** (D has been completely parsed);
23) **return** R;

Figure 4. Algorithm to search XTrie.

Algorithm MATCH−SUBSTRING ($ST, \mathcal{B}, r, \ell$)
Input: ST is the substring-table of an XTrie index. $\mathcal{B}$ is a 2-dim. integer-array. r refers to the first row in ST that corresponds to some substring that is matched at level ℓ.
Output: Set of matching XPEs.
1) Initialize R to be empty;
2) **while** ($r \neq 0$) **do**
3) $r' = ST[r].ParentRow$;
4) Initalize $match = false$;
5) **if** ($r' == 0$) **then**
6) **if** ($\ell \in ST[r].RelLevel$) **then**
7) $\mathcal{B}[r, \ell] = 1$;
8) **if** ($ST[r].NumChild == 0$) **then**
9) $match = true$;
10) **else**
11) **if** ($\exists\, \ell' \in [1, \ell - 1]$ such that $\ell - \ell' \in ST[r].RelLevel$ and $\mathcal{B}[r', \ell'] = ST[r].Rank$) **then**
12) $\mathcal{B}[r, \ell] = 1$;
13) **if** ($ST[r].NumChild == 0$) **then**
14) $match$ = PROPAGATE−UPDATE($ST, \mathcal{B}, r, \ell$);
15) **if** ($match$) **then**
16) Insert the id. of the XPE corrp. to row r into R;
17) $r = ST[r].Next$;
18) **return** R;

Figure 5. Algorithm to process a matched substring.

substrings, which are formed by the concatenation of some suffix of $label(N)$ and t, by using the max-suffix pointer in Step 10. For each end-tag t encountered (corresponding to some start-tag at level ℓ), the run-time information $\mathcal{B}$ is updated by resetting $\mathcal{B}[r, \ell]$ to 0 for all rows r (Step 18), and the search node is re-initialized to its previous location before the tag t was encountered (Step 19). This is achieved by using an array $Node$ to keep track of the location of the search node at each document level (Step 12).

Algorithm MATCH−SUBSTRING (Figure 5) is invoked when a substring s (matching at level ℓ) is detected. The algorithm checks for non-redundant matchings of s, updates the run-time information $\mathcal{B}$, and returns the identifiers of all the matching XPEs that have s as their last substring. More specifically, the algorithm iterates through each instance of s in ST (i.e., each row in the linked list associated with s) to check for non-redundant matchings of s. There are two scenarios for the instance of the matching substring (say $s_{i,j}$) corresponding to row r. For the special case where $s_{i,j}$ is a *root substring* (Steps 5-9), if its positional constraint is satisfied (Step 6), then the matching is a partial matching (and obviously non-redundant, since it is a root substring) and $\mathcal{B}[r, \ell]$ is updated to 1. If, in addition, $s_{i,j}$ is a leaf substring, then we have a matching of p_i (Step 9). For the general case where $s_{i,j}$ is a *non-root substring* (Steps 10-14), if there is a non-redundant matching of $s_{i,j}$ (Step 11), then $\mathcal{B}[r, \ell]$ is updated to 1. If, in addition, $s_{i,j}$ is a leaf substring, then Algorithm PROPAGATE−UPDATE is called to update the run-time information array $\mathcal{B}$ and check for a matching of p_i. We should point out that, since we are not interested in finding multiple matches of the same XPE, we should eliminate unnecessary processing and checking in MATCH−SUBSTRING for XPEs that have already been matched. This can be easily achieved by using a bit-mask (consisting of one bit per XPE); we have omitted details of this additional filtering step from Figure 5 for simplicity of presentation.

Algorithm PROPAGATE−UPDATE (depicted in Figure 6) is used to update $\mathcal{B}$ whenever a non-redundant subtree-matching of some non-root substring ($s_{i,j}$ matching at level ℓ corresponding to row r in ST) is detected. The algorithm iterates through each matching of its parent substring (at level $\ell' \in [\ell'_{min}, \ell'_{max}]$) and updates its $\mathcal{B}$ entry if the matching forms a non-redundant matching of $s_{i,j}$. If this matching is also a subtree-matching for the parent substring of $s_{i,j}$ (Step 12), then there are two cases to consider. If the parent substring is a root substring (Step 13), then we have found a matching of p_i; otherwise, we recurse the update propagation of the $\mathcal{B}$ entries for the ancestor substrings of $s_{i,j}$ as well (Step 16). The algorithm returns $true$ if a matching of p_i has been detected; otherwise, if it is possible to have multiple matchings of the parent substring of $s_{i,j}$ (i.e., $relLevel(s_{i,j}) = [\ell_{min}, \infty]$ for some ℓ_{min}), then to avoid any subsequent redundant matchings of descendants of $s_{i,j}$, the algorithm updates the $\mathcal{B}$ entries of all the earlier matchings of $s_{i,j}$ (Steps 18 to 20), and returns $false$.

The space requirement of the XTrie index is dominated by the total number of substrings in $\mathcal{P}$; that is, the space complexity is $O(\sum_{i=1}^{|\mathcal{P}|} |p_i|)$, where $|p_i|$ denotes the num-

Algorithm PROPAGATE-UPDATE ($ST, \mathcal{B}, r, \ell$)
Input: ST is the substring-table of an XTrie index.
$\mathcal{B}$ is a 2-dimensional integer-array. r refers to a row in ST that corresponds to some substring s of p for which there is a subtree-matching of s at level ℓ.
Output: Returns $true$ if there is a matching of p; $false$ otherwise.

$r' = ST[r].ParentRow$;
$[\ell_{min}, \ell_{max}] = ST[r].RelLevel$;
if ($\ell_{max} == \infty$) **then**
    $[\ell'_{min}, \ell'_{max}] = [1, \ell - \ell_{min}]$;
else
    $[\ell'_{min}, \ell'_{max}] = [\ell - \ell_{min}, \ell - \ell_{min}]$;
Initialize $match = false$;
Initialize $\ell' = \ell'_{max}$;
while ($match == false$) **and** ($\ell' \in [\ell'_{min}, \ell'_{max}]$) **do**
    if ($\mathcal{B}[r', \ell'] == ST[r].Rank$) **then**
        $\mathcal{B}[r', \ell'] = \mathcal{B}[r', \ell'] + 1$;
        if ($\mathcal{B}[r', \ell'] == ST[r'].NumChild + 1$) **then**
            if ($ST[r'].ParentRow == 0$) **then**
                $match = true$;
            else
                $match$ = PROPAGATE-UPDATE($ST, \mathcal{B}, r', \ell'$);
    $\ell' = \ell' - 1$;
if ($match == false$) **and** ($\ell_{max} == \infty$) **then**
    for $i = 1$ **to** $\ell - 1$ **do**
        if ($\mathcal{B}[r, i] > 0$) **then** $\mathcal{B}[r, i] = ST[r].NumChild + 1$;
return $match$;

Figure 6. Algorithm to update $\mathcal{B}$ and detect complete matchings of XPE.

ber of the substrings in the simple decomposition of p_i. To analyze the time complexity, let P denote the length of the longest root-to-leaf path in the trie T, L denote the maximum length of a linked list in ST, and H denote the maximum height of a substring-tree. The complexity of Algorithm PROPAGATE-UPDATE is $O(H\ L_{max})$. Since Algorithm MATCH-SUBSTRING makes at most L calls to Algorithm PROPAGATE-UPDATE, the complexity of Algorithm MATCH-SUBSTRING is $O(L\ H\ L_{max})$. For each start-tag in the input document, Algorithm SEARCH makes at most P calls to Algorithm MATCH-SUBSTRING; thus, the complexity of processing each start-tag is $O(P\ L\ H\ L_{max})$.

We conclude this section by briefly describing an optimized variant of XTrie, which we referred to as *Lazy XTrie*. In contrast to above variant of XTrie (referred to as *Eager XTrie*), which probes the substring-table ST for every matching substring detected in the input document, Lazy XTrie postpones the probing of ST such that the substring-table is only probed for a matching substring s if s appears as a leaf substring in some XPE; otherwise, for a matching non-leaf substring s, Lazy XTrie only updates information about the level at which s is matched in the input document. Thus, Lazy XTrie minimizes the number of unnecessary index probes at the expense of a slightly higher cost for each probe due to the additional processing required to check for matchings of the ancestor substrings of the matched leaf substring. The details of Lazy XTrie are given in [5].

5. Related Work

There has been various work on the filtering of data using "flat patterns" in the form of conjunctions of simple predicates on data attributes, including research on rule/trigger processing systems [9, 11] and publish-subscribe systems [1, 10, 14]. In contrast, our paper focuses on filtering XML documents based on tree patterns (based on XPath expressions), which demands more sophisticated indexing techniques, since tree patterns consist of both data contents as well as structure. The only work that is closely related to ours is the XFilter index which is also designed for filtering XML documents with XPath expressions [2]. While our XTrie index is based on decomposing tree patterns into collections of substrings (i.e., sequences of element names) and indexing them using a trie, XFilter essentially treats each tree pattern as a set of finite state automata, with each automaton responsible for the matching of some path in the tree pattern. The collection of automata for all the tree patterns are indexed using a hash table on the single element names (i.e., automata transitions).

XTrie is more space-efficient than XFilter since the space cost of XTrie is dominated by the number of substrings in each tree pattern, while the space cost of XFilter is dominated by the number of element names in each tree pattern. By indexing on substrings instead of single element names, the substring-table entries in XTrie are also probed less often than the hash table entries in XFilter. Furthermore, while XTrie ignores partial matchings of tree patterns that are redundant, XFilter keeps tracks of all instances of partially matched tree patterns, which results in more processing overhead.

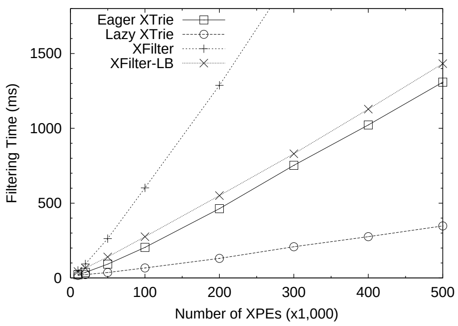
6. Experimental Evaluation

To determine the effectiveness of XTrie, we compare its performance against XFilter. Our results indicate that XTrie is between 2 to 4 times faster than XFilter for single-path XPEs[1].

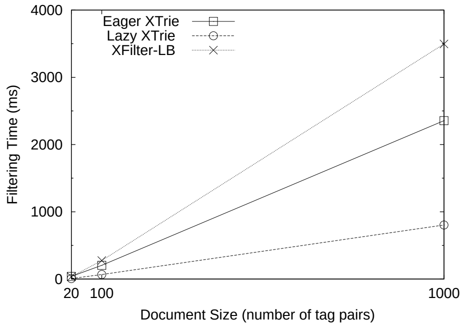
XML Documents. Similar to [2], we used the NITF (News Industry Text Format) DTD [7] to generate our XML document data set. The NITF DTD (version 2.5) contains 123 elements with 513 attributes. Our data set of XML documents is generated using IBM's XML Generator tool [8]. We generated three sets of 250 XML documents with similar characteristics. These sets correspond to different sizes of document: small, medium and large, with an average of 20, 100, and 1000 pairs of tags, respectively.

XPath Expressions. We implemented an XPath expression generator that takes a DTD as input and creates a set of valid XPath expressions (with no duplicates) based on the

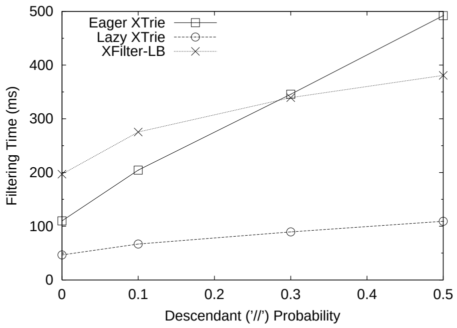
[1]We did not compare with tree-structured XPEs because the XFilter paper [2] focuses on single-path XPEs.



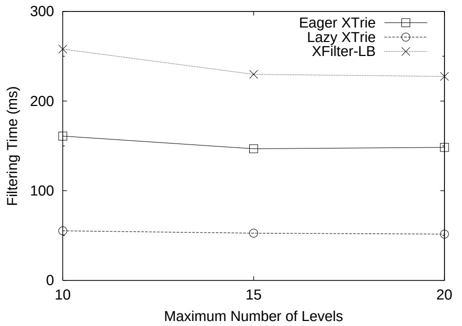
(a) Varying P ($L = 20, p_w = 0.1, p_d = 0.1, p_b = 0, \theta = 0$).



(b) Varying document length (P=$100k$, L=20, p_w=0.1, p_d=0.1, p_b=0, θ=0).



(c) Varying p_d (P=$100k$, L=20, p_w=0.1, p_b=0, θ=0).



(d) Varying L (P=$100k$, p_w=0.1, p_d=0.1, p_b=0, θ=0).

Figure 7. Experimental Results.

following set of six input parameters. The parameter P controls the cardinality of the set of indexed XPEs (ranging from $10,000$ to $500,000$). The parameter L controls the "depth" of the XPEs in terms of the maximum number of levels (ranging from 10 to 30). The parameter p_w (p_d) controls the probability (ranging from 0 to 0.5) of having a wildcard "$/*$" (descendant "$//$") operator at each node. The parameter p_b controls how "bushy" are the XPE-trees of the XPEs (ranging from 0 to 0.1); a value of 0 will generate only single-path XPEs, while a higher value will increase the number of branches in the XPE-trees. The parameter θ (ranging from 0 to 1) controls the skewness of the Zipf distribution [19] used for selecting element names, where a value of 0 corresponds to a uniform distribution and a higher value corresponds to a more skewed distribution.

Algorithms. We compare the performance of four algorithms: (1) XFilter, (2) XFilter with "list balance" optimization [2], which is denoted by $XFilter\text{-}LB$, (3) Eager XTrie, and (4) Lazy XTrie. Note that we did not apply the prefiltering optimization [2] to XFilter because this optimization is orthogonal to the index approach, and is applicable to XTrie as well. All the algorithms were implemented in C++ and compiled using GNU C++ version 2.95.3. Experiments were conducted on a Sun Ultra-250 with 512 MB of main memory running Solaris 2.7. All the index structures were resident in main-memory for all the experiments. For each input XML document, we measured the total filtering time which includes the CPU time to parse the input document, probe and update the index, and report the matched expressions. Our performance metric for each category of documents (small, medium, or large) is the average filtering time over the set of 250 XML documents for that category. We used the SAX parser of the Apache Foundation [3] for parsing XML documents. The average times for parsing a small, medium, and large document were 2.8 ms, 11.9 ms, 105.3 ms, respectively.

6.1. Experimental Results

Our experimental results are shown in Figure 7, where the base case uses the following parameter values: medium data set, $P = 10,000$, L=20, p_w=0.1, p_d=0.1, p_b=0, and θ=0.

Figure 7(a) compares the scalability of the algorithms as a function of P, the size of the set of indexed XPEs. The results show that the filtering time increases almost linearly with P, with Lazy XTrie being the fastest algorithm, which outperforms XFilter-LB by a factor of between 2 to 4. Eager XTrie performs slightly better than XFilter-LB, and XFilter performs the worst. Note that since the performance of XFilter is always much worse than XFilter-LB, we omit

XFilter from subsequent graphs. Figure 7(b) compares the scalability of the algorithms as a function of the size of the XML documents (in terms of the number of tag-pairs). The results clearly show that the filtering time increases linearly with the document size for all the algorithms.

Figure 7(c) shows that increasing the probability of descendant operators in the XPEs (i.e., p_d) increases the filtering time of all the algorithms. For the XTrie algorithms, this is because having more descendant operators in a XPE is likely to result in a larger number of shorter substrings in its simple decomposition, which not only increases the number of entries in the substring-table but also leads to more matchings in the trie (due to shorter substrings). For the XFilter-LB algorithm, having more descendant operators in the XPEs translates to more instances of partially matched expressions thereby resulting in more processing overhead.

Finally, Figure 7(d) compares the effect of the "depth" of the XPEs on the performance of the filtering algorithms. The graphs show that the performance of all the algorithms improves slightly as the depth of the XPEs increases. This is because tree patterns with longer "branches" are more selective resulting in fewer matches. More experimental results are given in [5].

We also compared the memory usage of both XTrie and XFilter, and our experimental results indicate that XTrie is more space efficient. For instance, for the experiment in Figure 7(a) with $500,000$ XPEs, XTrie required approximately 18 MB of memory while XFilter required 26 MB.

7. Conclusions

In this paper, we have proposed a novel index structure, termed XTrie, that supports the efficient filtering of streaming XML documents based on XPath expressions. Our XTrie index offers several novel features that make it especially attractive for large-scale publish/subscribe systems. First, the XTrie is designed to support effective filtering based on complex XPath expressions (as opposed to simple, single-path specifications). Second, our XTrie structure and algorithms are designed to support both ordered and unordered matching of XML data. Third, by indexing on sequences of XML element names (i.e., substrings) organized in a trie structure and using a sophisticated matching algorithm, the XTrie is able to both reduce the number of unnecessary index probes as well as avoid redundant matchings, thereby providing extremely efficient filtering. Our experimental results over a wide range of XML document and XPath expression workloads have clearly demonstrated the benefits of our approach, showing that our XTrie index consistently outperforms earlier approaches by wide margins.

Acknowledgements. We would like to thank Mehmet Altinel and Mike Franklin for helping us to understand the details of XFilter.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching Events in a Content-based Subscription System. In *Proc. of ACM PODC*, pages 53–61, Atlanta, GA, May 1999.

[2] M. Altinel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of VLDB*, pages 53–64, Sept. 2000.

[3] Apache. *Xerces C++ Parser*. http://xml.apache.org, 2001.

[4] A. Carzaniga, D. Rosenblum, and A. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Trans. on Computer Systems*, 19(3):332–383, August 2001.

[5] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. Technical report, Bell Labs., June 2001.

[6] T. I. Corporation. Intel NetStructure XML Accelerators. http://www.intel.com/netstructure/products/xml_accelerators.htm, 2000.

[7] R. Cover. *The SGML/XML Web Page*. http://www.oasis.open.org/cover/sgml-xml.html, Dec. 1999.

[8] A. Diaz and D. Lovell. *XML Generator*. http://www.alphaworks.ibm.com/tech/xmlgenerator, Sept. 1999.

[9] E. N. Hanson and M. Chaabouni and C.-H. Kim and Y.-W. Wang. A Predicate Matching Algorithm for Database Rule Systems. In *Proc. of ACM SIGMOD*, pages 271–280, Atlantic City, NJ, May 1990.

[10] F. Fabret, H. Jacobsen, F. Llirbat, K. Ross, and D. Shasha. Filtering Algorithms and Implementations for Very Fast Publish/Subscribe Systems. In *Proc. of ACM SIGMOD*, pages 115–126, Santa Barbara, California, May 2001.

[11] E. N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable Trigger Processing. In *Proc. of IEEE ICDE*, pages 266–275, Sydney, Austrialia, March 1999.

[12] D. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3, chapter 6.3. Addison Wesley, second edition, 1998.

[13] D. Megginson. *SAX: A Simple API for XML*. http://www.megginson.com/SAX/.

[14] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the Web. In *Proc. of ACM SIGMOD*, pages 437–448, Santa Barbara, California, May 2001.

[15] B. Segall, D. Arnold, J. Boot, M. Henderson, and T. Phelps. Content Based Routing with Elvin4. In *AUUG2K*, Canberra, Australia, June 2000.

[16] W3C. *Document Object Model (DOM) Level 1 Specification (Second Edition), Version 1.0.* http://www.w3.org/TR/REC-DOM-Level-1/.

[17] W3C. *XML Path Language (XPath) 1.0*. http://www.w3.org/TR/xpath, November 1999.

[18] W3C. *Extensible Markup Language (XML) 1.0, 2nd Edition*. http://www.w3.org/TR/REC-xml/, October 2000.

[19] G. Zipf. *Human Behaviour and Principle of Least Effort*. Addison-Wesley, Cambridge, Massachusetts, 1949.