The VLDB Journal (2002) 11: 354–379 / Digital Object Identifier (DOI) 10.1007/s00778-002-0077-6

Efficient filtering of XML documents with XPath expressions

Chee-Yong Chan, Pascal Felber*, Minos Garofalakis, Rajeev Rastogi

Bell Laboratories, Lucent Technologies, 600 Mountain Ave., NJ 07974, USA;
{cychan, pascal, minos, rastogi}@research.bell-labs.com

Edited by Alon Y. Halevy. Received: December 15, 2001 / Accepted: June 1, 2002
Published online: December 13, 2002 –

Abstract. The publish/subscribe paradigm is a popular model for allowing publishers (i.e., data generators) to selectively disseminate data to a large number of widely dispersed subscribers (i.e., data consumers) who have registered their interest in specific information items. Early 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. The emergence of XML as a standard for information exchange on the Internet has led to an increased interest in using more expressive subscription mechanisms (e.g., based on XPath expressions) that exploit both the structure and the content of published XML documents. Given the increased complexity of these new data-filtering mechanisms, the problem of effectively identifying the subscription profiles that match an incoming XML document poses a difficult and important research challenge. 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, we believe, 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 elements 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.

Keywords: Data dissemination – Document filtering – Index structure – XML – XPath

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 [6]. 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 [6], and Elvin [18], 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) [21] 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 [20], 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 [7]). 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."

* *Present address*: Institut EURECOM, Sophia Antipolis, France;e-mail: Pascal.Felber@eurecom.fr

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 complex tree-structured patterns with one or more wildcards, 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.

1.1 Our contributions

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 [14], to implement XML data filtering as the XML document is parsed. This is in contrast to the alternative DOM parsing interface [19], 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 scheme scales well to high volumes of XPEs and complex documents, and consistently outperforms XFilter by significant margins (factors of up to one or two orders of magnitude).

1.2 Roadmap

The remainder of this paper is organized as follows. In Sect. 2, we give an overview of the XPath language and discuss both the unordered and ordered matching mode for XPEs. Section 3 discusses our methodology for decomposing complex XPEs into *substrings* for effective indexing. In Sect. 4, we present our novel XTrie index structure and algorithms. Sect. 5 discusses two optimized variants of XTrie: the first variant is optimized to further reduce the number of unnecessary index probes, and the second variant is optimized for the special case where the indexed XPEs are simple, single-path expressions (rather than arbitrary trees). Section 6 compares XTrie against related work. In Sect. 7, we present the results of an extensive experimental study comparing the various variants of XTrie against the XFilter index [2]. Finally, we conclude in Sect. 8.

2 Background

In this section, we first present an overview of the XPath language for specifying path expressions over XML documents [20], followed by a discussion of the two modes of matching (unordered and ordered) for XPath expressions.

2.1 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

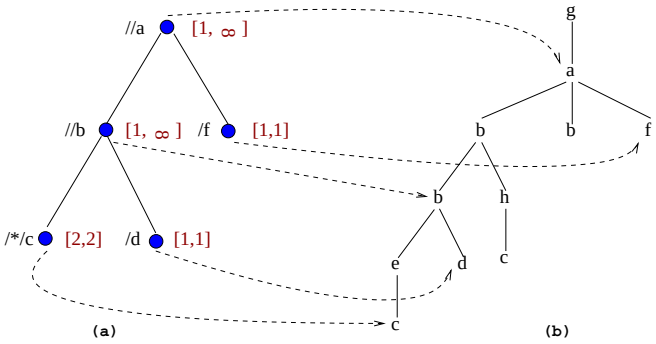
are enclosed by "[" and "]" symbols. The predicates can be applied to the text or the attributes of the addressed elements, and may also include other path expressions. Any relative paths in a predicate expression are evaluated in the context of the element nodes addressed in the location step at which they appear. For example, the XPE $/a[b[@x \geq 100]/c]/*/d$ specifies a tree-structured 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 of at least 100.

In this paper, we focus on a fragment of the XPath language commonly referred to as *tree patterns*, which represents a significant and useful fragment of XPEs; in fact, tree patterns have been used extensively in the literature as a natural and intuitive means for specifying tree-structured constraints in XML and LDAP applications [3,4,17,16]. A tree pattern is 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 "*/"), and 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-structured representation of an XPE as an *XPE-tree*. As an example, Fig. 1a depicts the XPE-tree of the expression $p = //a[.//b[*/c][./d]]/f$. Note that in Fig. 1a, the child node for $*/c$ precedes the child node for d since the former precedes the latter in the expression for p. Our tree patterns allow for predicates comparing element/attribute values against constants, but not join predicates involving a comparison of two path expressions; we believe that our tree patterns probably capture the key features of XPEs that will prove most useful in data-dissemination applications.

2.2 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 pre-order traversal of T, denoted by $v \prec_{pre} v'$, if v is visited before v' in a pre-order traversal of T. Given an XML document tree, we associate each node d in the tree with a *level number*, denoted by $level(d)$, where $level(d) = 1$ if d is the root element of the document; otherwise, $level(d) = level(d') + 1$, where d' is the parent node of d. For example, in Fig. 1b, the element "f" is at level 3. In addition, given an XPE-tree T, we associate each node t in T 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 exactly k, denoted by $relLevel(t) = [k, k]$. Figure 1a shows the relative-level annotations for the nodes in our example XPE-tree.

Consider an XPE-tree T 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 mapping, referred to as a *node mapping* (denoted by f), from the nodes in T to the nodes in D such that: (1) for each node t_i in T, t_i matches at $f(t_i)$ in D; and (2) for each child node t_j of a node t_i in T, $f(t_j)$ is a descendant of $f(t_i)$ in D such that $level(f(t_j)) - level(f(t_i)) \in relLevel(t_j)$. In other words, our definition ensures that both: (1) the labels of individual elements in the XPE are matched in the document; and (2) the *positional constraints* specified in the XPE are met. As an example, consider the XPE-tree T of $p = //a[.//b[*/c]/d]/f$ in Fig. 1a, and the XML document tree D in Fig. 1b, where T matches D with the node mapping indicated by the set of dashed arrows from the nodes in T to those in D.



a **b**

Fig. 1a,b. Unordered and ordered matchings. **a** XPE-tree of $p = //a[.//b[*/c]/d]/f$. **b** Example XML tree of a document D

In addition to the model of unordered matchings, XPath also allows the order of matching to be explicitly specified. A key reason for this is that the preservation of ordering constraints is a basic requirement in several application domains (e.g., document processing). Consider again the XPE-tree in Fig. 1a 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 $p' = //a[.//b/*[following\text{-}sibling::d]/c]/f$ [1]. Referring again to Fig. 1b, if the positions of the two subtrees rooted at e and d 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 t_i in T, $t_j \prec_{pre} t_k$ in T if and only if $f(t_j) \prec_{pre} f(t_k)$ and $f(t_i)$ is the least common ancestor node of $f(t_j)$ and $f(t_k)$. Condition (2) basically ensures that sibling substrings matched along distinct branches in the XML document tree.

To simplify the presentation, we discuss unordered and ordered matchings of XPEs in terms of their XPE-trees. Abstractly, for ordered matching, the order in which the child branches of each XPE-tree node are matched is the same as the left-to-right ordering depicted in the XPE-tree, whereas for unordered matching this order is immaterial. Thus, for the remainder of the paper, we focus mainly on XPEs that are formed using the basic operators (i.e., child-operator "/" and descendant-operator "//") and view their XPE-trees as ordered (unordered) trees for ordered (resp., unordered) matchings. *Hybrid matchings* of XPEs, involving both unordered as well as ordered node matchings, are also discussed later in the paper.

[1] Other order-related operators in XPath include $following$::, $preceding$::, and $preceding\text{-}sibling$:: [20].

Table 1. Notation

Symbol	Description
$\mathcal{P}$	Set of XPEs being indexed.
$\mathcal{S}$	Set of distinct substrings from the simple decompositions of all the XPEs in $\mathcal{P}$.
$\|p_i\|$	Number of substrings in the simple decomposition of p_i.
$s_{i,j}$	j^{th} substring in a decomposition of XPE p_i.
$L_{\max}$	Maximum number of levels in XML document.
$label(N)$	Label of trie node N in XTrie.
$\alpha(N)$	Substring pointer of trie node N in XTrie.
$\beta(N)$	Max-suffix pointer of trie node N in XTrie.

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 explain how the substrings resulting from such a decomposition can be organized into *substring-trees* for effective matching over streaming XML documents. We also define several important concepts for matching based on substring-trees that play a key role in our XTrie indexing structure and matching algorithms.

We begin by summarizing (in Table 1) some of the key notational conventions used in our discussion in the remainder of the paper. We provide detailed definitions of the parameters in the text once all the relevant concepts are presented. Additional notation will be introduced when necessary.

3.1 Substring decompositions and substring-trees

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. As an example, consider the XPE $p = /a/b[c/d//e][g//e/f]//*/*/e/f$ whose XPE-tree is depicted in Fig. 2a. 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 "/".

Let $P = < p_1, p_2, \cdots, p_n >$ be a sequence of paths in the XPE-tree of an XPE p that satisfies all the following three properties: (1) for each p_i in P, the concatenation of the element names of all the nodes along p_i is a substring of p (denoted by s_i); (2) p_i precedes p_j in P iff the last node in p_i precedes the last node in p_j in the pre-order traversal of the XPE-tree of p; and, (3) each node in the XPE-tree of p is contained in at least one path in P. We refer to the sequence of substrings $< s_1, s_2, \cdots, s_n >$ corresponding to P as a *substring decomposition of p*. 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 2a 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.

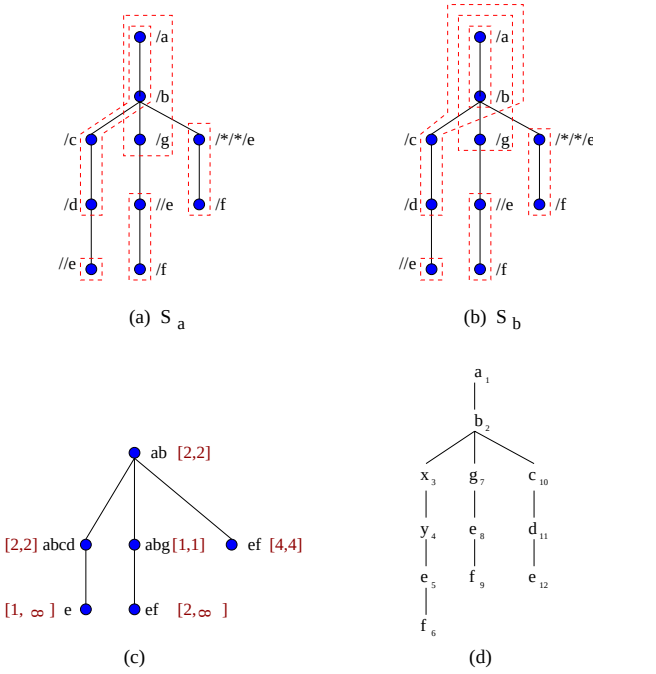


Fig. 2. Substring decompositions. **a** $S_a = < abcd, e, abg, ef, ef >$ **b** $S_b = < ab, abcd, e, abg, ef, ef >$ **c** Substring-tree for S_b **d** Example XML document tree D

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 minimal decompositions. On the other hand, using only a minimal decomposition for an XPE can result in problems when checking for an unordered match under our SAX-based parsing model for XML documents. As an example, consider again the minimal decomposition S_a of an XPE p in Fig. 2a with $s_1 = abcd$, $s_2 = e$, $s_3 = abg$, $s_4 = ef$, $s_5 = ef$, and the XML document tree D in Fig. 2d, where the numeric subscripts denote the order in which the document elements are seen through the SAX parsing interface. Clearly, p matches D in the unordered matching model. A matching algorithm for p that relies on p's substring decomposition needs to match the substrings in that decomposition in some partial order that enables the *positional constraints* between each matching substring and its "parent" to be checked as the document nodes are streaming by through the SAX-based document parser. For example, to correctly detect a matching of s_6, the element e must be matched at exactly three levels below where the element b in $abcd$ (or abg) is matched. The prob-

lem with this example is that the matching of ef (after f_6 is parsed in D) occurs before the matchings of both $s_1 = abcd$ and $s_3 = abg$ and, therefore, there is no matching ocurrence of either of these substrings to enable checking the positional constraints for ef. The key problem here, of course, is that ab appears only as a prefix of substrings s_1 and s_3, and not as an explicit substring in the decomposition of p.

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 Fig. 2b 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. In addition, note that, for a single-path XPE, its simple decomposition is equal to its minimal decomposition.

The substrings of the simple decomposition of an XPE p can be organized into a unique rooted tree, referred to as the *substring-tree* of p. Let $S = < s_1, s_2, \cdots, s_n >$ denote the simple decomposition of p corresponding to the sequence of paths $P = < p_1, p_2, \cdots, p_n >$. Then, the substring-tree of p is constructed as follows:

1. The *root substring* is s_1.
2. For each substring $s_i \in S$, $i > 1$, the *parent substring* of s_i is s_j (or equivalently, s_i is the *child substring* of s_j), if the last node of p_j (among all the paths in P) is the nearest ancestor node of the last node of p_i.
3. The ordering among sibling substrings is based on their ordering in S.

As an example, Fig. 2c shows the substring-tree for the simple decomposition S_b of p depicted in Fig. 2b. We define the *rank of a substring* s_i to be equal to k if s_i is the k^{th} child of its parent substring; the rank of the root substring is 1. A substring that has no child substrings is called a *leaf substring*. For example, in Fig. 2c, the ranks of s_4 and s_5 are 2 and 1, respectively; and the leaf substrings are s_3, s_5, and s_6.

We now extend the notion of relative level that was defined for nodes in XPE-trees to substrings. Abstractly, the relative level of a substring s refers to the range of possible differences in levels between the last elements of s and its parent substring in a matching. More formally, let $S = < s_1, s_2, \cdots, s_n >$ be the substring decomposition of an XPE p corresponding to the sequence of paths $P = < p_1, p_2, \cdots, p_n >$. Consider a substring s_i in S (with parent substring s_j), and let V denote the set of nodes in p_i that are not in p_j. Let x denote $\sum_{v_k \in V} \ell_k$, where $relLevel(v_k) = [\ell_k, u_k]$. Then, the *relative level* of s_i is defined to be at least x, denoted by $relLevel(s_i) = [x, \infty]$, if $\max_{v_k \in V}\{u_k\} = \infty$; otherwise, it is defined to be exactly x, denoted by $relLevel(s_i) = [x, x]$. Figure 2c shows the relative-level annotations for the nodes in the substring tree for the simple decomposition S_b.

3.2 Matching with substrings

Consider an XML document tree D and an XPE p with XPE-tree T and simple decomposition $< s_1, s_2, \cdots, s_n >$ corresponding to the sequence of paths $P = < p_1, p_2, \cdots, p_n >$. Suppose p matches D; i.e., there is a node mapping f from the nodes in T to those in D. We can extend the definition of matching for XPE nodes to substrings as follows: s_i *matches* at a node d in D (or there is a *matching* of s_i at d in D) if $f(v)$ matches at d in D, where v is the last node of p_i. For notational convenience, we use $f(s_i) = v$ to denote a matching of s_i at node v under the node mapping f. We say that there is a matching of s_i at level ℓ in D if s_i matches at some node at level ℓ in D. Clearly, to fully match p, we need to find a matching for each of the substrings of p such that the *positional constraint* defined by p between each substring and its parent is satisfied.

As the nodes in D are parsed in a pre-order traversal (by the SAX parser), the ordered matching of p in D also progresses incrementally following a pre-order traversal of the substring-tree of p such that each substring s_i is matched before s_{i+1}. Thus, to determine if p matches D, we need to keep track of the *partial matchings* of p in D. However, since we are interested only in whether or not p matches D and not in the actual number of match occurrences, partial matchings of p 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. Let p' be a new XPE that is equivalent to p except that p' is formed using only the first i paths in P, for some $i \in [1, n]$. We say that there is a *partial matching* of substring s_i at a node d in D if p' matches D such that the last node of p_i matches at d in D. We represent a partial matching by its node matching f that maps nodes from T to nodes in D. It follows that we have a *(complete) matching* of p in D if there is a partial matching of s_n at some node in D.

A partial matching of s_i at node d in D, where d is the k^{th} node in the pre-order traversal of D, is defined to be a *redundant matching* if for each XML document D' (that is equivalent to D for the first k nodes) that matches p under a mapping f with $f(s_i) = d$, there exists an alternative mapping f' that also defines a complete matching of p but with $f'(s_i) \prec_{pre} d$. As an example, consider again Fig. 1, where the simple decomposition of p is $< a, b, c, bd, af >$. Note that each of the partial matchings indicated by the dashed arrows in Fig. 1 is a non-redundant matching. There are, however, two redundant matchings (which are not explicitly shown): (R1) the partial matching of substring c at the c node under the node h, and (R2) the partial matching of substring b at the second b node under the node a.

Informally, a partial matching of a substring s_i is redundant if there already exists a preceding partial matching of s_i such that ignoring the later partial matching would not affect the correctness of deciding whether or not p matches D. Since we are not interested in the actual number of occurrences for a match, the efficiency of filtering documents with XPEs can be improved by detecting and ignoring redundant substring matches so as to reduce the overhead of book-keeping operations to maintain such partial matchings. To enable efficient detection of redundant matchings, we introduce the notion of *subtree-matchings*.

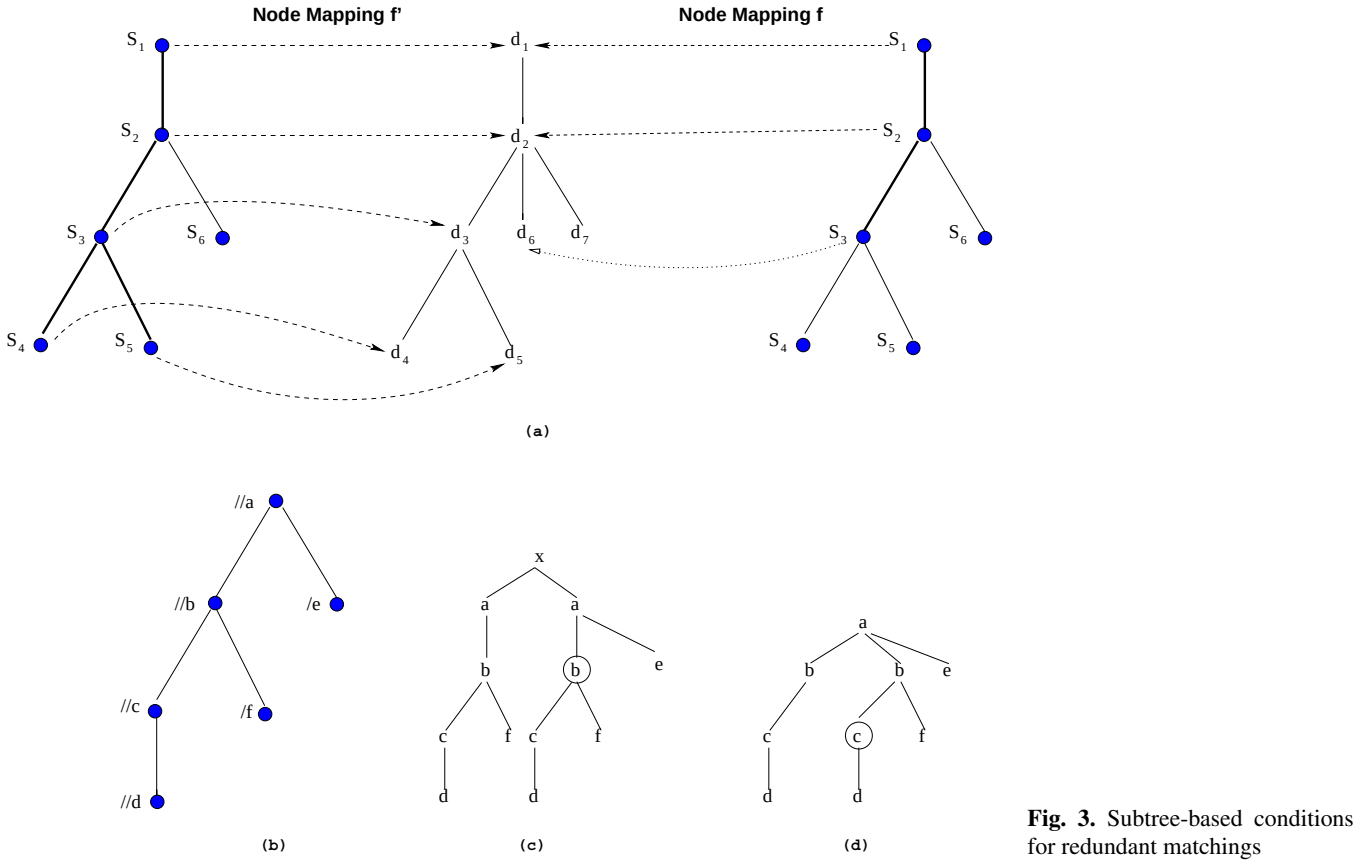


Fig. 3. Subtree-based conditions for redundant matchings

A node mapping f is said to define a *subtree-matching* of s_i if f defines a partial matching of *each descendant* of s_i; that is, f actually captures a matching that includes the entire XPE subtree rooted under s_i. As an example, consider again the substring-tree in Fig. 2c, and assume that a partial matching of the substring ef (whose parent substring is abg) has just been detected. This implies that there is a subtree-matching for each of the following four substrings: $abcd$, abg, e and ef itself. Subtree-matchings provide a useful operational means to capture redundant matchings. Referring to the two redundant matchings (R1) and (R2) in Fig. 1, the partial matching of substring c in (R1) is redundant because there already exists a subtree-matching of its ancestor substring b, while the partial matching of substring b in (R2) is redundant because there already exists a subtree-matching of the substring b itself.

Thus, a convenient approach to detect redundant matchings is to keep track of subtree-matchings for the various substrings. More formally, a partial matching of s_i (defined by a mapping f) is *redundant* if there exists another partial matching of s_i (defined by a mapping f') such that: (1) $f'(s_i) \prec_{pre} f(s_i)$; and, (2) there exists an ancestor substring s_a of s_i such that: (a) $f'(s_a) = f(s_a)$; and (b) f' defines a subtree-matching of the child substring of s_a whose subtree contains s_i.

An example to illustrate the above subtree-based conditions for redundant matchings is depicted in Fig. 3a, where two node mappings, f and f', are shown for matching a substring-tree with (six substrings) to an XML document (with seven nodes). Suppose that the node d_6 in the XML document has just been parsed and it matches the substring s_3. By our subtree-based conditions, the partial matching of s_3 at d_6 (defined by f) is redundant because there already exists an earlier partial matching of s_3 at d_3 (defined by f') which is part of a subtree-matching (in this example, a subtree-matching of s_3 itself), where both f' and f map s_2, the parent substring of s_3, to the same node d_2. To understand why we require the condition that $f'(s_a) = f(s_a)$, consider the XPE-tree and XML document tree in Figs. 3b and c, respectively. Without this condition on s_a, the partial matching of substring b to the circled b node in Fig. 3c would have been incorrectly considered to be redundant since there is subtree-matching of substring b at an earlier b node. The XML document tree in Fig. 3d illustrates why we need the condition that there be a subtree-matching at the *child* substring of s_a (as opposed to at some *descendant* substring of s_a) whose subtree contains s_i. If the weaker condition is used, then the partial matching of substring c to the circled c node in Fig. 3c would have been incorrectly regarded as redundant since there is a subtree-matching of substring c at an earlier c node.

The above subtree-based conditions essentially detect redundant matchings based on information about earlier substrings that have already been matched. However, since there could be redundant matchings that can only be detected through information about "yet-to-be-matched" substrings, our subtree-based conditions are in fact only *sufficient* for redundant matchings. For example, consider the XPE p and XML document D in Figs. 4a and b. Note that the partial matching of the substring b at the leaf b node is actually redundant, but this would not be detected by our subtree-based conditions since there is no subtree-matching of the substring

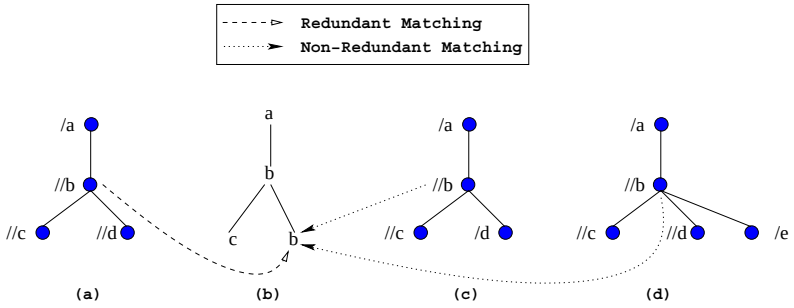


Fig. 4. Redundant and non-redundant matchings

a. The reason for this redundancy is because the b substring has only one unmatched child substring d that has a prefix of // which means that subsequent matchings of b will not affect the overall matching outcome. This type of redundant matching clearly depends on the contents of the "yet-to-be-matched" substrings; for instance, if the prefix of substring d had been / as in Fig. 4c, or if the b substring had another child substring that is prefixed with / (e.g., Fig. 4d), then the partial matching of the substring b at the leaf b node would have been non-redundant. Since our subtree-based approach relies only on information about substrings that have already been matched, it is incapable of detecting such redundant matchings. Detecting such redundancies is more complex and would require more elaborate book-keeping operations. Our proposed XTrie index structure is based on the simple approach of detecting redundant matchings using earlier subtree-matchings so as to minimize the book-keeping overhead.

4 The XTrie indexing scheme

In this section, we present our novel XTrie indexing scheme for filtering XML documents based on XPEs. We first describe the XTrie index structure and matching algorithm for the ordered matching model (Sects. 4.1 to 4.4), and then explain how our approach can be extended to handle unordered matchings (Sect. 4.5) and hybrid matchings (Sect. 4.6). To simplify the presentation, our discussion in this section focuses mostly on XPEs that: (1) do not refer to any attribute names or text data; and (2) do not involve Boolean combinations of XPEs. Section 4.7 then explains how our approach can deal with Boolean combinations of XPEs (i.e., composite XPEs) with attributes and/or text data.

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* [13] (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, and briefly discuss the maintenance issues for the XTrie index.

4.2 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 $s_{i,j}$ (denoting the j^{th} substring in the decomposition of p_i). 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 symbol $r_{i,j}$ to denote the row in ST that corresponds to 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 $\mathcal{S}$[2].

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 $Next = 0$.

4.3 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

[2] Note that clustering the rows by XPEs in ST simplifies its maintenance. If updates are very infrequent, an alternative scheme is to cluster rows that correspond to the same substring together. In this way, ST becomes more space-efficient since we can effectively eliminate the *next* pointers required for the linked lists.

to node N (the label of the root node is the empty string). The construction of T ensures that: (1) for each $s \in \mathcal{S}$, there is a unique node N in T such that $label(N) = s$; and (2) 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) determined 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, $\alpha(N) = 0$.
- 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.

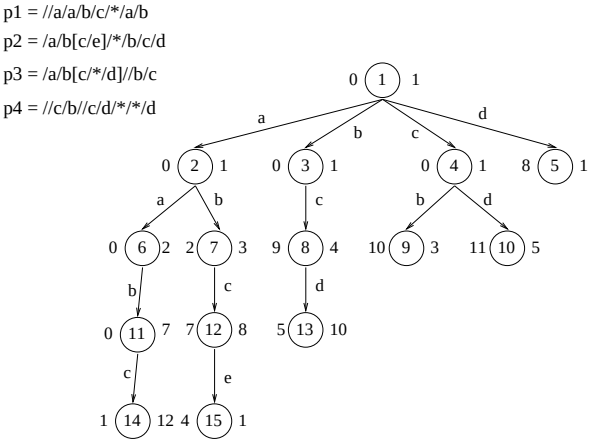
Example 4.1 : Figure 5 depicts the XTrie index structures for a set of four XPEs $\mathcal{P} = \{p_1, p_2, p_3, p_4\}$ (shown in the figure), where their respective simple decompositions are as follows: $S_1 = < aabc, ab >, S_2 = < ab, abce, bcd >, S_3 = < ab, abc, d, bc >$, and $S_4 = < cb, cd, d >$. The number within each trie node N in Fig. 5a represents the node's identifier, and the values of $\alpha(N)$ and $\beta(N)$ are shown to the left and right of N, respectively. Fig. 5b depicts the corresponding substring-table with the rows clustered in the order of the XPEs in $\mathcal{P}$. □

4.3.1 Maintaining the XTrie index

The maintenance of the XTrie index structure when XPEs are inserted into or deleted from it is rather straightforward, with the exception of maintaining the max-suffix pointers in the trie which is slightly more involved. One approach to efficiently maintain these pointers is to build an auxiliary suffix trie structure T_{rev} on the set of reversed substrings so that for each node N in T, there exists an unique node N' in T_{rev} such that $label(N) = reverse(label(N'))$. By enhancing T_{rev} with special node pointers $\gamma(.)$ so that $\gamma(N')$ points to its associated node in T (i.e., $\gamma(N') = N$ iff $label(N) = reverse(label(N'))$), the max-suffix pointer value of a node N in T can be determined easily by traversing T_{rev} using $reverse(label(N))$: if N' is the last node reached by $reverse(label(N))$ in T_{rev}, and N'' is the closest ancestor node of N' that has a non-null value for $\gamma(.)$, then $\beta(N)$ is given by $\gamma(N'')$. The details of the auxiliary structure and the XTrie maintenance algorithms can be found in Appendix A.

4.4 The XTrie matching algorithm

Our XTrie indexing scheme is designed to support on-line filtering of streaming XML data and is based on the SAX event-based interface that reports parsing events. Fig. 7 depicts 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.



a

	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

b

Fig. 5. XTrie example. **a** Trie T. **b** Substring-table ST

The basic idea of our 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 ST only stores static information on the XPE substrings, we need to maintain some additional dynamic run-time information to ensure that we check only for non-redundant matchings. Of course, we also need to appropriately update this dynamic information as the parsing of the document progresses and new substring matchings are discovered.

Our XTrie matching algorithm maintains run-time information using two arrays $\mathcal{B}$ and $\mathcal{C}$ each of which is a two-dimensional array 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[3].

[3] Note that the value for the $L_{\max}$ parameter can be set to a sufficiently large value by exploiting a priori knowledge of incoming XML documents. In the event that the $L_{\max}$ value is exceeded at run-time, more space for the run-time information can be dynamically reallocated.

- The first array $\mathcal{B}$ is an integer-array such that $\mathcal{B}[r_{i,j}, \ell] = n$, $n > 0$, if there is a non-redundant matching of $s_{i,j}$ (represented by a node mapping f) 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., f defines a subtree-matching of the $(n-1)^{th}$ child substring of $s_{i,j}$). Intuitively, $\mathcal{B}[r_{i,j}, \ell]$ records the rank of the *next child subtree of $s_{i,j}$ that we need to match* for this non-redundant occurrence of $s_{i,j}$ at level l. Thus, 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 $s_{i,1}$. 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 $n + 1$, $n \geq 1$, when there is 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 start-tag at level ℓ is parsed.
- The second array $\mathcal{C}$ is a bit-array that is used to ensure that sibling substrings match along *distinct* branches (as defined in Sect. 2) for an ordered matching. Each entry $\mathcal{C}[r_{i,j}, \ell]$ corresponds to a matching of the substring $s_{i,j}$ at level ℓ, and is initialized with a value of 0. Whenever the value of $\mathcal{B}[r_{i,j}, \ell]$ is incremented to some value $k > 1$, indicating that a subtree-matching of the $(k-1)^{th}$ child substring of $s_{i,j}$ has been detected, $\mathcal{C}[r_{i,j}, \ell]$ is set to 1. $\mathcal{C}[r_{i,j}, \ell]$ is then reset back to 0 right before the next document node at level ℓ is to be parsed (i.e., when an end-tag corresponding to a start-tag at level ℓ is parsed in the input XML document). Informally, a value of $\mathcal{C}[r_{i,j}, \ell] = 1$ indicates that the nodes parsed in the input document are along the same branch as the one that matched the $(k-1)^{th}$ child substring of $s_{i,j}$; therefore, any matching of the k^{th} child substring of $s_{i,j}$ (with $s_{i,j}$ matching at level ℓ) detected during this period can not be considered a valid partial matching.

To understand how the arrays $\mathcal{B}$ and $\mathcal{C}$ are 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: (1) $\mathcal{C}[r_{i,k}, \ell']$ has a value of 0; (2) $\ell - \ell' \in relLevel(s_{i,j})$ (i.e., the positional constraint between $s_{i,j}$ and $s_{i,k}$ is satisfied); and (3) $\mathcal{B}[r_{i,k}, \ell'] \geq n$ (i.e., we have subtree matchings for at least the $n-1$ left-siblings of $s_{i,j}$ rooted at $s_{i,k}$). If, in addition, the value of $\mathcal{B}[r_{i,k}, \ell']$ is exactly n, then this partial matching is non-redundant; otherwise, we have already discovered a subtree matching for $s_{i,j}$, so the current matching is redundant and can safely be ignored. Note that since both $\mathcal{B}$ and $\mathcal{C}$ are large sparse arrays, their implementation can be optimized to minimize space (e.g., using linked lists)

As an example of how the $\mathcal{B}$ array is used to detect non-redundant matchings, consider the substring-subtree (consisting of substrings s_1 to s_8) in Fig. 6a, which shows a partial matching of s_5. A shaded node for s_i means that there is a partial matching of s_i; and for notational convenience, assume that the partial matching of s_i ($1 \leq i \leq 6$) is at some node at level ℓ_i of some XML document. The number to the right of each node s_i represents its $\mathcal{B}[s_i, \ell_i]$ value. For instance, in

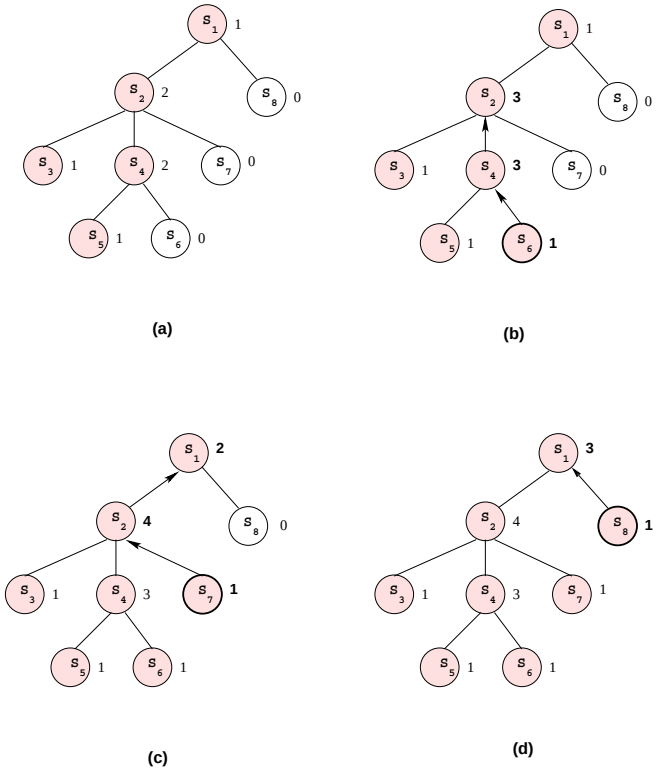


Fig. 6. Propagation of subtree-matchings

Fig. 6a, the $\mathcal{B}$ array value for s_2 is equal to 2 since only its first child substring (i.e., s_3) is part of a subtree-matching. Subsequently, when a partial matching of s_6 is detected (shown in Fig. 6b), it also trivially follows that there is a subtree-matching of s_6 since s_6 is a leaf substring. In order to correctly maintain the $\mathcal{B}$ array values, we need to propagate information about the subtree-matching of s_6 up to its parent substring (i.e., s_4) to indicate that a subtree-matching has been detected for its second child substring. This update propagation (indicated by an up arrow from s_6 to s_4 in Fig. 6b) therefore increments s_4's $\mathcal{B}$ array value by one to 3, which in turn indicates that there is a subtree-matching of s_4. Consequently, we need to further propagate the update upwards to s_2 and increment its $\mathcal{B}$ array value by one to 3. The update propagation stops at this point since there is no subtree-matching of s_2. Given the updated $\mathcal{B}$ array values in Fig. 6b, it is clear that a subsequent partial matching of s_4 would be considered redundant since $\mathcal{B}[s_2, \ell_2]$ is now greater than the rank of s_4. For a similar reason, a subsequent partial matching of either s_3, s_5, or s_6 is also considered redundant. Figures 6c and d, show how the $\mathcal{B}$ array values are updated after a partial matching of s_7 and s_8, respectively.

As a more concrete example to illustrate how the $\mathcal{B}$ array values are updated and used, Table 2 depicts an execution trace of the changes to the $\mathcal{B}$ array when matching XPE p against the XML document D in Fig. 1. The second column of the table describes the changes to the $\mathcal{B}$ array after processing the start tag indicated in the first column[4]. For instance, after the first c node in D is parsed, a partial matching of c, which is

[4] For simplicity, we have omitted showing the changes to $\mathcal{B}$ after the processing of end tags.

Table 2. Execution trace of changes to $\mathcal{B}$ array for the matching of p on D in Fig. 1

Start Tag	Changes to B array after processing start tag
g	
a	$B[a,2]=1$.
b	$B[b,3]=1$.
b	$B[b,4]=1$.
e	
c	$B[c,6]=1$, $B[b,4]=2$.
d	$B[bd,5]=1$, $B[b,4]=3$, $B[a,2]=2$, $B[b,3]=3$.
h	
c	Redundant matching of c since $B[b,3]$ is greater than the rank of c.
b	Redundant matching of b since $B[a,2]$ is greater than the rank of b.
f	$B[af,3]=1$, $B[a,2]=3$, Complete matching of p.

also a subtree-matching, is detected; and this is propagated to its to its parent substring b resulting in updates to both $B[c,6]$ and $B[b,4]$.

4.4.1 Details of matching algorithm

Our XTrie SEARCH algorithm (depicted in Fig. 7) begins by initializing the search node N to be the root node of the trie T (Step 6). 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 17. 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 substrings, which are formed by the concatenation of some suffix of $label(N)$ and t, by using the max-suffix pointer in Step 11. 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 19), and the search node is re-initialized to its previous location before the tag t was encountered (Step 20). This is achieved by using an array $Node$ to keep track of the location of the search node at each document level (Step 13).

Algorithm MATCH-SUBSTRING (Fig. 8) 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 (to indicate that we can start looking for matchings of child subtrees). 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 10-15), 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 arrays $\mathcal{B}$ and $\mathcal{C}$, and check for a matching of the full XPE 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 Fig. 8 to simplify the presentation.

Algorithm PROPAGATE-UPDATE (depicted in Fig. 9) is used to implement such "update propagations" and correctly update both $\mathcal{B}$ and $\mathcal{C}$ 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 $s_{i,j}$'s parent substring (at level $\ell' \in [\ell'_{min}, \ell'_{\max}]$) and updates its $\mathcal{B}$ and $\mathcal{C}$ entries 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 13), then there are two cases to consider. If the parent substring is a root substring (Step 14), then we have found a matching of p_i; otherwise, we recurse the update propagation of the $\mathcal{B}$ and $\mathcal{C}$ entries for the ancestor substrings of $s_{i,j}$ as well (Step 17). 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 19 to 22), and returns $false$.

4.4.2 Space and time complexity

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 number of substrings in the simple decomposition of p_i. To analyze the search-time complexity, let P denote the length of the longest root-to-leaf path in the trie T, let L denote the maximum length of a linked list in ST (i.e., the number of distinct occurrences of any substring), and let H denote the maximum height of a substring-tree. The worst-case time 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 worst-case complexity of processing each start-tag in an input document is $O(P\ L\ H\ L_{\max})$. Finally, it is easy to see that processing an end-tag takes $O(|ST|)$ time; thus, the overall (worst-case) time complexity of processing each tag in an input XML document is $O\left(\max\{P\ L\ H\ L_{\max}\ ,\ |ST|\}\right)$.

```
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 B be a |ST| × L_max integer-array with all values initialized to 0;
4)  Let C be a |ST| × L_max bit-array with all values initialized to 0;
5)  Initialize ℓ = 0;  // ℓ is the current document level
6)  Initialize N to be the root node of T;  // N is the current trie node
7)  repeat
8)      if (a start-tag t is parsed in D) then
9)          ℓ = ℓ + 1;
10)         while ((there is no edge labeled "t" from N) and
            (N is not the root node of T)) do
11)             N = β(N);
12)         if (there is an edge labeled "t" from N to N′ in T) then
13)             Node[ℓ] = N′; N = N′;
14)             while (N′ is not the root node) do
15)                 if (α(N′) > 0) then
16)                     R = R ∪ MATCH-SUBSTRING (ST, B, C, α(N′), ℓ);
17)                 N′ = β(N′);
18)     else if (an end-tag is parsed in D) then
19)         Reset B[i, ℓ] to 0 for i = 1 to |ST|;
20)         Node[ℓ] = root node of T;
21)         ℓ = ℓ − 1;
22)         Reset C[i, ℓ] to 0 for i = 1 to |ST|;
23)         N = Node[ℓ];
24) until (D has been completely parsed);
25) return R;
```

Fig. 7. Algorithm to search XTrie

```
Algorithm MATCH-SUBSTRING (ST, B, C, r, ℓ)
Input: ST is the substring-table of an XTrie index. B is a 2-dimensional integer-array.
       C is a 2-dimensional bit-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 ≠ 0) do
3)      r′ = ST[r].ParentRow;
4)      Initalize match = false;
5)      if (r′ == 0) then
6)          if (ℓ ∈ ST[r].RelLevel) then
7)              B[r, ℓ] = 1;
8)              if (ST[r].NumChild == 0) then
9)                  match = true;
10)     else
11)         if (∃ ℓ′ ∈ [1, ℓ − 1] such that ℓ − ℓ′ ∈ ST[r].RelLevel,
12)         B[r′, ℓ′] == ST[r].Rank, and C[r′, ℓ′] == 0) then
13)             B[r, ℓ] = 1;
14)             if (ST[r].NumChild == 0) then
15)                 match = PROPAGATE-UPDATE (ST, B, C, r, ℓ);
16)     if (match) then
17)         Insert the id. of the XPE corresponding to row r into R;
18)     r = ST[r].Next;
19) return R;
```

Fig. 8. Algorithm to process a matched substring

```
Algorithm PROPAGATE-UPDATE (ST, B, C, r, ℓ)
Input: ST is the substring-table of an XTrie index. B is a 2-dimensional integer-array.
       C is a 2-dimensional bit-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.
1)  r' = ST[r].ParentRow;
2)  [ℓ_min, ℓ_max] = ST[r].RelLevel;
3)  if (ℓ_max == ∞) then
4)      [ℓ'_min, ℓ'_max] = [1, ℓ − ℓ_min];
5)  else
6)      [ℓ'_min, ℓ'_max] = [ℓ − ℓ_min, ℓ − ℓ_min];
7)  Initialize match = false;
8)  Initialize ℓ' = ℓ'_max;
9)  while (match == false) and (ℓ' ∈ [ℓ'_min, ℓ'_max]) do
10)     if (B[r', ℓ'] == ST[r].Rank) then
11)         B[r', ℓ'] = B[r', ℓ'] + 1;
12)         C[r', ℓ'] = 1;
13)         if (B[r', ℓ'] == ST[r'].NumChild + 1) then
14)             if (ST[r'].ParentRow == 0) then
15)                 match = true;
16)             else
17)                 match = PROPAGATE-UPDATE (ST, B, C, r', ℓ');
18)     ℓ' = ℓ' − 1;
19) if (match == false) and (ℓ_max == ∞) then
20)     for i = 1 to ℓ − 1 do
21)         if (B[r, i] > 0) then
22)             B[r, i] = ST[r].NumChild + 1;
23) return match;
```

Fig. 9. Algorithm to update run-time information arrays and detect complete matchings

4.5 Dealing with unordered matching

We now describe how our XTrie indexing scheme can handle an unordered matching model. For convenience, we refer to the two variants of XTrie as *ordered XTrie* and *unordered XTrie*.

Recall that in the ordered matching model, since the child substrings of each parent substring are matched in a specific order (based on their ranks), it is sufficient, for each parent substring, to keep track of only its "leftmost" child substring for which a subtree-matching has not been detected using the integer-array $\mathcal{B}$. However, in the unordered matching model, since child substrings can be matched in any order, it becomes necessary to explicitly keep track of the *subset of child substrings* for which subtree-matchings have been detected. Thus, the first main difference between the two variants of XTrie lies in the type of run-time information maintained in $\mathcal{B}$. Specifically, for unordered XTrie, $\mathcal{B}$ is a bitstring-array instead of an integer array, where each bitstring consists of $(w+1)$ bits, and w is the maximum number of child substrings over all substrings in an XPE-tree. For notational convenience, we number the bits in a bitstring from zero to w such that the leftmost bit is the 0^{th} bit, and the rightmost bit is the w^{th} bit. The bitstring values in $\mathcal{B}$ are initialized and updated as follows:

(1) For each ℓ, $\mathcal{B}[r_{i,j}, \ell]$ is initialized to all zero bits except for the leftmost $(k+1)$ bits which are all set to one, where k denotes the number of child substrings of $s_{i,j}$.
(2) The 0^{th} bit of $\mathcal{B}[r_{i,j}, \ell]$ is reset to 0 when a partial matching of $s_{i,j}$ at level ℓ is detected.
(3) The n^{th} bit of $\mathcal{B}[r_{i,j}, \ell], n > 0$, is reset to 0 when a subtree-matching of the n^{th} child substring of $s_{i,j}$ at level ℓ is detected.
(4) Each $\mathcal{B}[r_{i,j}, \ell]$ is re-initialized (as explained in (1)) when an end-tag (corresponding to a start-tag at level ℓ) is parsed.

It follows that, when there is a subtree-matching of $s_{i,j}$ at level ℓ, the value of $\mathcal{B}[r_{i,j}, \ell]$ is 0. The second main difference is that the $\mathcal{C}$ array is *not required* for unordered matching, since the matchings for two sibling substrings can in fact be along the same branch of the XML document.

The algorithms for unordered matching are very similar to those for order matching (shown in Figs. 7, 8, and 9) except that there is no need for the $\mathcal{C}$ array, and the operations/checks on the $\mathcal{B}$ array have to be modified accordingly based on the above discussion. (The detailed changes to our XTrie algorithms are straightforward and omitted here for the sake of brevity.)

4.6 Dealing with hybrid matching

We now discuss how our XTrie scheme can be extended to handle hybrid matchings involving a combination of ordered and unordered matchings. An example of an XPE that requires hybrid matching is $p = /a[b[following\text{-}sibling::c]][d[following\text{-}sibling::e]/f$, where the root element a has a set of five child elements $\{b, c, d, e, f\}$ such that b must precede c and d must precede e.

We first consider the simpler scenario where, for each non-leaf substring s of an XPE p, the matchings required for its child substrings are either completely unordered or com-

pletely ordered; we refer to s as an *unordered substring* and *ordered substring*, respectively. (Leaf substrings are considered ordered substrings.) To handle the matching of such XPEs, we need to enhance XTrie with the following two extensions. First, to indicate whether a substring is an unordered or ordered substring, one simple approach is to store the rows in the substring-table ST such that ST is partitioned into a block of consecutive rows for ordered substrings and another block of consecutive rows for unordered substrings. Second, since the type of run-time information needed for unordered and ordered matchings is different, instead of maintaining a single $\mathcal{B}$ array, we now need to use two smaller arrays, $\mathcal{B}_{ordered}$ and $\mathcal{B}_{unordered}$, for ordered and unordered substrings, respectively. Specifically, $\mathcal{B}_{ordered}$ is an $(N_{ordered} \times L_{\max})$ integer-array and $\mathcal{B}_{unordered}$ is an $(N_{unordered} \times L_{\max})$ bitstring-array, where $N_{ordered}$ and $N_{unordered}$ denote the number of ordered and unordered substrings, respectively. Similarly, $\mathcal{C}$ is now a smaller $(N_{ordered} \times L_{\max})$ bit-array to be used only for ordered substrings. The search algorithms need to update $\mathcal{B}_{ordered}$, $\mathcal{B}_{unordered}$, and $\mathcal{C}$ accordingly for ordered and unordered substrings.

We now briefly explain how the above approach can be further extended to handle the most general case of hybrid matchings, where a parent substring is allowed to have both unordered and ordered child substrings. The basic idea is to introduce additional *"dummy" nodes* to the XPE-tree so as to transform the complex case to the simpler case that we have just described. To illustrate our approach, consider again the earlier example with $p = /a[b[following\text{-}sibling::c]][d[following\text{-}sibling::e]/f$. For this example, we will create a modified XPE-tree by adding two dummy nodes π_{bc} and π_{de} (each with an empty string as its label), such that the root node a now has three child nodes: π_{bc}, π_{de}, and f, where π_{bc} is the parent node of b and c; and π_{de} is the parent node of d and e. The substring-tree for this modified XPE-tree consists of eight substrings: $s_1 = a$ is the root substring with three child substrings: $s_2 = a$, $s_5 = a$, and $s_8 = af$, where s_2 has two child substrings $s_3 = ab$ and $s_4 = ac$, and s_5 has two child substrings $s_6 = ad$ and $s_7 = ae$. Among these eight substrings, only the root substring is an unordered substring. Thus, by adding two dummy nodes, we have reduced the problem to the simpler scenario of ordered/unordered substrings that we have already addressed.

4.7 Attributes, text data, and composite XPath expressions

So far, our discussion of XTrie has been limited to XPEs that do not refer to any attributes or text data, and that are not *composite XPEs* (i.e., Boolean combinations of XPEs). In this section, we explain how XTrie can be easily extended to handle attributes, text data, and composite expressions.

To handle XPEs with attributes, we just need to extend the substring-table ST with an additional column, *Attribute*, which is a pointer to a list of attributes (including any predicates) associated with the elements in a substring. For example, consider the XPE $p = /a[@name][@address]/b[@cost \leq 500]/c[d]$, where element a must have two attributes "$name$" and "$address$", and element b must have an attribute "$cost$" with a value of no more than 500. The simple decomposition of p consists of three substrings: $s_1 = ab$, $s_2 = abc$, and $s_3 = abd$. Let r_1, r_2, and r_3 denote the rows in ST that correspond to s_1, s_2, and s_3, respectively. Then, the *Attribute* value of row r_1 points to a linked list consisting of two entries with information about the attributes associated with the elements a and b. (Note that this information will not be repeated in rows r_2 and r_3 to avoid redundancy.) In addition, since both elements c and d are not associated with any attributes, their values for *Attribute* is a null value representing an empty attribute list. By keeping track of the attributes (and their values if any) associated with the elements as they are parsed in an input XML document, the additional constraints on attributes can be easily verified for each matching substring. Thus, a matching for a substring s is considered to be a partial matching of s if all the attribute constraints associated with s are also satisfied.

Note that predicates that involved text values are handled in a similar manner as described for predicates involving attributes; where the substring-table is extended with an additional column, *Text*, which is a pointer to a list of predicates on the text values associated with the elements in a substring. Essentially, in XTrie, we used the SAX parser to generate a single event for each start-element tag which consists of the element name, all the attributes specified in the start-element tag, and any text value enclosed after the start-element tag. In this way, any predicates associated with an element can be checked after its start-element event is reported by the SAX parser.

To handle composite XPEs, a simple and efficient approach is to split each composite XPE into its constituent simple XPEs and index these simple XPEs. The matching of each composite XPE can be checked by examining the matching results of its constituent XPEs after the document has been completely parsed. (A similar approach is also adopted by XFilter [2].) For example, the composite XPE $p = //a/b$ OR $//c/d$ can be split into two basic XPEs $p_1 = //a/b$ and $p_2 = //c/d$ so that there is a matching of p if and only if there is a matching of p_1 or p_2.

Furthermore, absolute path expressions in predicate expressions that do not involve join expressions are easily supported by XTrie. For example, the XPE $p = //a/b[/c/d = 2]//e$ is treated as two XPEs $p_1 = //a/b//e$ and $p_2 = /c[d = 2]$.

5 Optimizations for XTrie

In this section, we describe two optimizations for XTrie. Our first optimization is based on a "lazy" XTrie variant that aims to further reduce the number of unnecessary index probes. Our second optimized XTrie variant tries to improve the performance for the special case where all the indexed XPEs are single-path XPEs. For simplicity, we shall discuss these optimizations under the ordered matching model.

5.1 Lazy XTrie

The XTrie variant that we have presented so far (referred to as *Eager XTrie*) probes the substring-table ST for every matching substring detected in the input document. Our optimized *Lazy XTrie* variant (described in this section) tries to reduce

the number of unnecessary index probes by postponing the probing of the substring-table ST so that ST is probed for a matching substring s only if s appears as a *leaf substring* in some XPE; otherwise, Lazy XTrie only updates information about the level at which s is matched in the input document. In this section, we explain the main differences between the lazy and eager variants of XTrie; the details of the search algorithm for Lazy XTrie are given in Appendix B.

An important consequence of this optimization is that the order in which substring matchings are processed in Lazy XTrie follows a bottom-up approach as opposed to Eager XTrie which follows the pre-order traversal of the XPE's substring-tree. To illustrate this difference, consider again the substring-tree in Fig. 6. For Eager XTrie, the order of the partial matchings for the substrings follow the sequence s_1, s_2, $\cdots$, s_8. On the other hand, for Lazy XTrie, it first processes the matching of the leaf substring s_3 and then propagates upwards to process the matchings of substrings s_2 and s_1 (if they exist). Next, it detects and processes the matching of the second leaf substring s_5 followed by an upward propagation to process the matching of s_4 (if it exists). The remaining substrings (which are all leaf substrings) are detected and processed in the order s_6, s_7, and s_8. Thus, Lazy XTrie does not always immediately check if a matched substring constitutes a partial matching, but only does so in a bottom-up manner when the matched substring is a leaf substring. This difference in operation introduces a number of structural and algorithmic differences between Eager and Lazy XTrie.

Structurally, Eager and Lazy XTrie are almost equivalent except for the following three differences. First, since Lazy XTrie only probes the substring-table when the matched substring s is some leaf substring, we need to "remember" all the matched substrings that have been detection prior to the matching of a leaf substring. For this book-keeping, we maintain an additional data structure, denoted by $\mathcal{M}$, which is a $(|\mathcal{S}| \times L_{\max})$ bit-array such that $\mathcal{M}[i, \ell]$ is set to 1 if and only if the substring s is matched at level ℓ of the input document, where $i \in [1, |\mathcal{S}|]$ represents the identifier of s. For ease of access to the substring identifiers, we explicitly store the substring identifiers in a new attribute, denoted by SID, in the substring-table such that $ST[r_{i,j}].SID$ is the identifier of the substring $s_{i,j}$. Second, in order to ensure that the substring-table is only probed for a matching leaf substring, we need to distinguish between leaf and non-leaf substrings. This is achieved by simply negating the values of $\alpha(N)$ in the trie if $label(N)$ does not correspond to a leaf substring. Finally, unlike Eager XTrie, where there are $|\mathcal{S}|$ linked lists in ST (with one list per distinct substring in $\mathcal{S}$); Lazy XTrie has only $|\mathcal{S}_{leaf}|$ linked lists in ST, where $\mathcal{S}_{leaf} = \{s \in \mathcal{S} \mid \text{s is a leaf substring in some XPE}\}$; with one linked list for each substring in $\mathcal{S}_{leaf}$ such that a row $r_{i,j}$ in ST belongs to a linked list for substring s if and only if $s_{i,j}$ is a leaf substring of p_i and $s_{i,j} = s$. Thus, many of the rows in ST would not belong to any linked list at all.

Algorithmically, the main search algorithm for Lazy XTrie is almost equivalent to that for Eager XTrie (in Fig. 7) except that it now records occurrences of all matched substrings and probes the substring-table only when the matched substring is a leaf substring. However, checking if a matched substring s constitutes a partial matching in Lazy XTrie is more complex than in Eager XTrie due to the bottom-up approach of processing matched substrings in Lazy XTrie. In contrast to Eager XTrie, where the $\mathcal{B}$ array information about the ancestor substrings of a matched substring s have already been properly initialized to be used for processing s, this is not necessarily the case in Lazy XTrie. In particular, if s is the first child substring of its parent substring s', then the $\mathcal{B}$ array information on s' has not been initialized and we first need to determine that there is a partial matching of s' itself, which might in turn lead to further propagation up the chain of ancestor substrings.

5.2 XTrie for single-path XPEs

We now present an optimized variant of XTrie for the special case where all the indexed XPEs are single-path XPEs. Since single-path XPath expressions are simpler, we believe that they could be typical in applications where, for example, the users do not have sophisticated requirements. Moreover, since matching single-path XPath expressions is more efficient than tree-structured ones, even in a general scenario where only some of the indexed XPEs are single-path ones, it might be more efficient to separately index the single-path and tree-structured patterns. By exploiting the simple structure of such XPEs, both the data structures as well as the algorithms of XTrie can be further fine-tuned. In the following, we focus on the optimized Lazy XTrie for single-path XPEs; the details of the single-path-optimized Eager XTrie are given in Appendix C. Note that, for single-path XPEs, ordered and unordered matchings are equivalent.

Lazy XTrie for single-path XPEs differs from Lazy XTrie for tree-structured XPEs in the following ways. First, since each substring in a single-path XPE has at most one child substring, the substring-table ST for single-path XPEs is simpler than that for tree-structured XPEs. Specifically, each row in ST (corresponding to some substring $s_{i,j}$) is a 4-tuple $(RelLevel,\ RootSubstr,\ SID,\ Next)$, where $RelLevel$, SID, and $Next$ are defined as earlier (Sects. 4.2 and 5.1), and $RootSubstr$ is a single bit that is set to 1 if and only if $s_{i,j}$ is the root substring of p_i (i.e., $j = 1$). Note that the attributes $Rank$ and $NumChild$, which are necessary for tree-structured XPEs, are redundant for single-path XPEs. Furthermore, for each XPE, by ordering a parent substring before its child substring in ST, the attribute $ParentRow$ need not be explicitly stored, since each parent substring is always located in the row preceding the row of its (only) child substring. Second, since each parent substring has exactly one child substring, there is no need to maintain the run-time information arrays $\mathcal{B}$ and $\mathcal{C}$.

The main search algorithm for single-path XPEs is equivalent to that for tree-structured XPEs (shown in Fig. 21) except that the $\mathcal{B}$ and $\mathcal{C}$ arrays are not needed; the detailed matching algorithms are depicted in Fig. 10. Algorithm `LAZY-MATCH-SUBSTRING` for single-path XPEs is almost equivalent to that for tree-structured XPEs (in Fig. 21) except that it involves fewer parameters. Algorithm `MATCH-SUBSTRING-SUB` for single-path XPEs, however, is clearly significantly simpler than its general-case counterpart. For each substring $s_{i,j} \in \mathcal{S}_{leaf}$ (corresponding to row r in ST) matching at level ℓ, Algorithm `LAZY-MATCH-SUBSTRING` is invoked to check whether or not this matching forms a complete matching of p_i. The algo-

Algorithm LAZY-MATCH-SUBSTRING (ST, $\mathcal{M}$, r, ℓ)
Input: ST is the substring-table of an XTrie index. $\mathcal{M}$ is a 2-dimensional bit-array.
r refers to the first row in ST that corresponds
to some leaf substring that matches at level ℓ.
Output: Set of matching XPEs.
1) Initialize R to be empty;
2) **while** ($r \neq 0$) **do**
3) $match$ = MATCH-SUBSTRING-SUB (ST, $\mathcal{M}$, r, ℓ);
4) **if** ($match$) **then**
5) Insert the id. of the XPE corresponding to row r into R;
6) $r = ST[r].Next$;
7) **return** R;

Algorithm MATCH-SUBSTRING-SUB (ST, $\mathcal{M}$, r, ℓ)
Input: ST is the substring-table of an XTrie index. $\mathcal{M}$ is a 2-dimensional bit-array.
r refers to a row in ST that corresponds to some substring $s_{i,j}$
that matches at level ℓ.
Output: Returns $true$ if there is a complete matching of p_i; $false$, otherwise.
1) Initialize $match = false$;
2) **if** ($ST[r].RootSubstr$) **then** //r corresponds to a root substring
3) **if** ($\ell \in ST[r].RelLevel$) **then**
4) $match = true$;
5) **else** //r corresponds to a non-root substring
6) $r' = r - 1$;
7) $parentSid = ST[r'].SID$;
8) Initialize $\ell' = \ell - \ell_{\min}$, where $ST[r].RelLevel = [\ell_{\min}, \ell_{\max}]$;
9) **while** ($match == false$) **and** ($\ell' > 0$) **and** ($\ell - \ell' \in ST[r].RelLevel$) **do**
10) **if** ($\mathcal{M}[parentSid, \ell']$) **then**
11) $match$ = MATCH-SUBSTRING-SUB (ST, $\mathcal{M}$, r', ℓ');
12) $\ell' = \ell' - 1$;
13) **return** $match$;

Fig. 10. Algorithm to process a matching substring in Lazy XTrie for single-path XPEs

rithm recursively looks for a matching of the parent substring of $s_{i,j}$ at level ℓ' that is consistent with the matching of $s_{i,j}$ (i.e., $\ell - \ell' \in ST[r].RelLevel$); the algorithm returns $true$ if and only if the root substring of p_i is finally matched, implying a complete matching of p_i.

6 Related work

Earlier work has proposed various approaches for the problem of filtering data using "flat patterns" in the form of conjunctions of simple predicates on data attributes. This includes research on rule/trigger processing systems [10,12] and publish-subscribe systems [1,11,15]. In contrast, our work focuses on filtering XML documents based on tree patterns (i.e., XPath expressions), which demands more sophisticated indexing techniques, since such patterns comprise both data content and 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. Each automaton is represented by a linked list of nodes, where each node represents a state in the automaton; and each link, which is labeled with an element name, represents a state transition. Note that each node has at most one out-going link labeled with an element name; and the collection of linked lists of nodes is indexed using a hash table on the element names (i.e., automata transitions) such that nodes whose incoming links share the same element name label are chained together in the hash table. Specifically, each hash table entry (corresponding to some element name t) consists of two linked lists: a dynamic *candidate-list* and a static *wait-list*. The candidate-list for element name t consists of nodes representing potential next states that are reachable by a transition on element name t, and it is initialized with states that are reachable from start states; the wait-list consists of all the remaining states reachable with a transition on element name t from non-start states. As the input XML document is parsed, the potential next-states of the automata are updated by copying nodes from the appropriate wait-lists to their corresponding candidate-lists or deleting nodes from the candidate-lists.

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. In the worst case, a candidate-list in XFilter can grow to a length of $|\mathcal{P}|2^L$ after an element at level L is parsed, where $|\mathcal{P}|$ is the total number of XPEs being indexed. We illustrate this exponential space complexity of XFilter with the following example. Consider a set of n XPEs $\mathcal{P} = \{p_1, p_2, \cdots, p_n\}$, where each of the n XPEs shares the same prefix expression given by "$//x[y = v_1]//x[y = v_2]// \cdots //x[y = v_m]$". The

prefix expression essentially consists of a sequence of m number of element x that are separated by descendant operators such that each element x is associated with a distinct predicate "$y = v_i$", where y is another element and each v_i is a distinct constant value. Therefore, for element x, its initial candidate list consists of n nodes (one node for each XPE), and its wait list consists of $n(m-1)$ nodes ($m-1$ nodes for each XPE); while for element y, its initial candidate list is empty, and its wait list consists of nm nodes (m nodes for each XPE). Consider an XML document tree D that consists of only a single path of element names all of which are labeled x. Since the first element x in D matches the first x element in all the XPEs (i.e., all the nodes in the candidate list of element x are matched), the candidate list for element x is updated by copying n nodes from its wait list (corresponding to the second element x in each XPE). Similarly, the candidate list for element y is updated by copying n nodes from its wait list (corresponding to the expression "$y = v_1$" in each XPE). Subsequently, when the second element x in D is parsed, it again matches all the nodes in the candidate list of element x (i.e., it matches both the first and second element x of each XPE); therefore, the candidate list for element x is updated by copying $2n$ nodes (corresponding to the second and third element x in each XPE) from the wait list of element x into its candidate list. Similarly, the candidate list for element y is updated by copying $2n$ nodes from its wait list (corresponding to both the expressions "$y = v_1$" and "$y = v_2$" in each XPE) thereby increasing its number of nodes to $3n$. At this point, each XPE p_i is associated with two nodes that correspond to the expression "$y = v_2$" in the candidate list for element y. It is important to note that these two nodes are not equivalent (and therefore the second copy is not redundant) because they are associated with different meta information; in particular, their values for the "$level$" attribute (which indicates the level at which the nodes should be matched) are 2 and 3. Thus, it follows that the candidate list for element y would have a total of $2^{i-1}n$ distinct nodes after the element x at the i^{th} level in D is parsed.

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 track of all instances of partially-matched tree patterns, which results in higher processing overheads.

7 Experimental study

To determine the effectiveness of XTrie, we have conducted exhaustive experiments with the various variants of the XTrie algorithm, as well as XFilter, under a wide range of XML document and XPath expression workloads. Note that our implementation of XFilter is based on the description in [2]. Our results indicate that XTrie scale well to high workloads and consistently outperforms XFilter by significant margins.

7.1 Testbed and methodology

Table 3. Characteristics of the default XML data sets (100 documents per set)

DTD	10 DTDs	NITF
Number of tags T:		
avg [min, max]	120.3 [85, 180]	108.5 [45, 132]
Number of levels:		
avg [min, max]	16.65 [5, 36]	18.11 [7, 24]

7.1.1 XML documents

For the experiments presented here, we have used two distinct data sets. The first one measures the effectiveness of the algorithms with multiple DTDs. This "aggregate" data set was created from 10 real-world DTDs used in major commercial applications. Among the 10 DTDs, the smallest DTD contains 77 elements with 1,377 attributes, while the largest DTD has 2,727 elements with 8512 attributes. Each DTD was equally represented in the data set, i.e., we generated $1/10$ of the XML documents and XPEs with each DTD.

The second data set was used to precisely measure how the different algorithms react to slight variations of the workload. We have used the News Industry Text Format (NITF) DTD[8], developed as a joint standard by news organizations and vendors worldwide, and supported by most of the world's major news agencies. The NITF DTD (version 2.5) contains 123 elements with 513 attributes. Note that the same DTD was used in [2] for XFilter's performance study.

Our data set of XML documents has been generated using IBM's XML Generator tool [9]. The NITF DTD and most of the other DTDs contain recursive structures, which can be nested to produce XML documents with arbitrary number of levels. We have generated sets of 100 XML documents with similar characteristics. The documents of the default data sets have approximately 100 tags and 20 levels.[5] The default data sets are used in all experiments, except when measuring the scalability of the filtering algorithms with respect to the size of the documents. For the latter experiment, we have generated additional data sets by varying the number of tags. Table 3 shows the main characteristics of our default XML document data sets.

7.1.2 XPath expressions

We implemented an XPath expression generator that takes a DTD as input and creates a set of valid XPath expressions based on the input parameters shown in Table 4. The parameter P controls the size of the set of indexed XPEs. The parameter L controls the maximal "depth" of the XPEs in terms of the maximum number of levels. The average depth is approximately equal to half the maximal depth. The parameters p_* and $p_{//}$, respectively, control the probabilities of having a wildcard "$/*$" or a descendant "$//$" operator at each node; therefore, the probability of the child operator "$/$" is given by $(1 - p_* - p_{//})$. The parameter p_λ controls how "bushy" are the XPE-trees of the XPEs; a value of 0 will generate only

[5] Note that in the context of data dissemination, documents are generally small. For instance, a typical XML document describing a stock quote has approximately 10 tags and 3 levels.

Table 4. Parameters in XPath expression generator

Parameter	Meaning	Values
P	Number of XPath expressions.	10K to 200K
L	Maximum number of levels in the XPEs.	8 to 20
p_*	Probability of having a wildcard "$/*$" operator.	$[0, 0.5]$
$p_{//}$	Probability of having a descendant "$//$" operator.	$[0, 0.5]$
p_λ	Probability of branching.	$[0, 0.1]$
θ	Skewness of element names.	$[0, 5]$

single-path XPEs, while a higher value will increase the number of branches in the XPE-trees. The parameter θ controls the skewness of the Zipf distribution [22] used for selecting element names. When set to 0, element names are randomly chosen according to a uniform distribution from the set of allowed elements. When set to positive values, the choice of the element names is skewed.

7.1.3 Algorithms

We compare the performance of the XFilter algorithms and the XTrie algorithms. We have implemented two variants of XFilter for single-path XPEs as described in [2]: the basic variant (referred to as XFilter) and the variant with the list balancing optimization (referred to as XFilter-LB). We have also added support for unordered matching of tree-structured XPEs to XFilter, without the list balancing optimization, according to the informal description in [2]. We have tested all variants of the XTrie algorithms: eager and lazy, single-path and tree-structured, ordered and unordered. Unless explicitly mentioned, we use for XTrie the simple decomposition introduced in Sect. 3 to split XPEs into substrings. 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.

We implemented the algorithms in C++ and compiled them using GNU C++ version 2.96. Experiments were conducted on a 1.5 GHz Intel Pentium 4 machine with 512 MB of main memory running Linux 2.4.2. All the index structures were resident in main-memory for all the experiments. For each experiment, the query evaluation time that we measured includes the CPU time to parse the input XML document, probe and update the index, and report matches to the application. XML file parsing was performed using the SAX parser of the Apache Foundation [5]. The average parsing time per document when no filtering takes place was approximately 6 ms for the default XML document data sets.

7.2 Experimental results

7.2.1 Scalability

We first compared the scalability of XTrie and XFilter with single-path XPEs by varying the number P of XPEs up to $200,000$. The results with both data sets (Figs. 11a and b) show that the filtering time of the XTrie algorithms increases linearly with the number of XPEs, with the lazy variant being 2–3 times faster than the eager variant. The performance of the XFilter algorithm decreases linearly with P when using the NITF DTD, and logarithmically with the 10 DTDs. This can be explained by the fact that the aggregate data set contains a large number (more than $5,000$) distinct element names. Therefore, the lists of candidates managed by the XFilter algorithm remain small and never exceeds about one tenth of P at any given time. With the NITF DTD, lists can become prohibitely long and, unsurprisingly, the list balancing optimization yield a much bigger performance improvement with that DTD. Note that the XTrie algorithms consistently outperforms XFilter by almost one order of magnitude.

We then compared the performance of the XTrie and XFilter algorithms with tree-structured XPEs (Figs. 12a and b). The filtering time of the XTrie algorithms increases linearly with the number of XPEs. Unlike with single-path XPEs, the lazy variants of the XTrie algorithms do not perform better than their eager counterparts. With the aggregate data sets, the eager variant performs slightly better, while the lazy variant has a thin edge with the NITF DTD. This can be explained by the fact that, since tree-structured expressions have several leaf substrings, the benefits of making fewer accesses to the substring table in the lazy variant are balanced – and sometimes even surpassed – by the higher costs of these accesses. Lazy XTrie is expected to perform better than Eager XTrie when leaf substrings occur infrequently in XML documents. The results also show that the unordered XTrie algorithms are less efficient that the ordered algorithms, because the unordered algorithms use more complex data structures and must keep track of a larger number of partial matches.

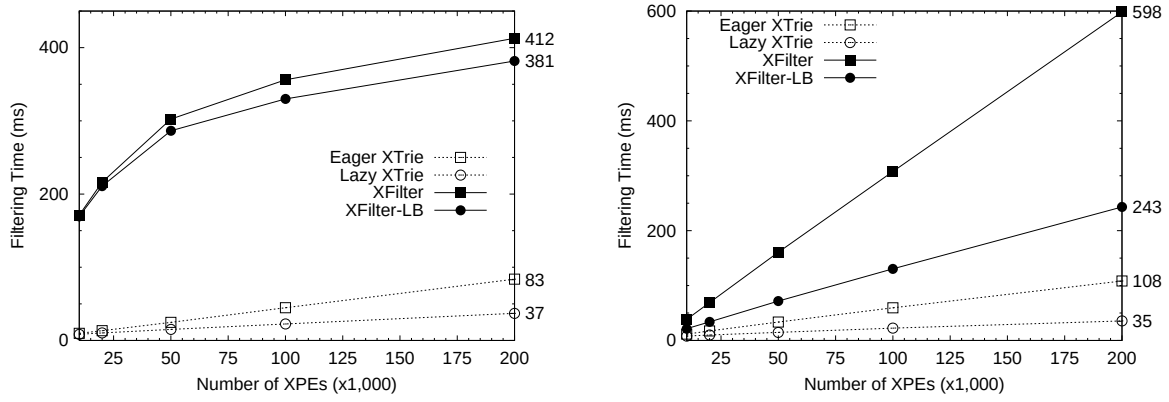
With the aggregate data set, the performance of the unordered XFilter algorithm degrades linearly with P. When using the NITF DTD, however, the performance of XFilter is extremely poor and degrades exponentially[6] with the number of XPEs. This gap between the aggregate and NITF data sets is analoguous to the behavior observed with single-path XPEs. With tree-structured XPEs, XFilter further suffers from the branching factor, which can dramatically increase the number nodes copied to the candidate lists.

Figures 13a and b, and 14a and b analyze the scalability of the filtering algorithms with respect to the average length of the XML documents for single-path and tree-structured XPEs.[7] It appears clearly that the processing time increases linearly with the number of tags for all algorithms, with both types of XPEs. Remarkably, it appears that the list balancing optimization of XFilter becomes useless and even penalizes performance with large XML documents. This can be explained by the fact that large documents use many of tags in a DTD, some of which correspond to short candidate lists. The use of large documents indirectly yields an optimization similar to list balancing, but without the extra overhead.

In the rest of this section, unless explicitly mentioned, we present the results obtained with the NITF DTD.

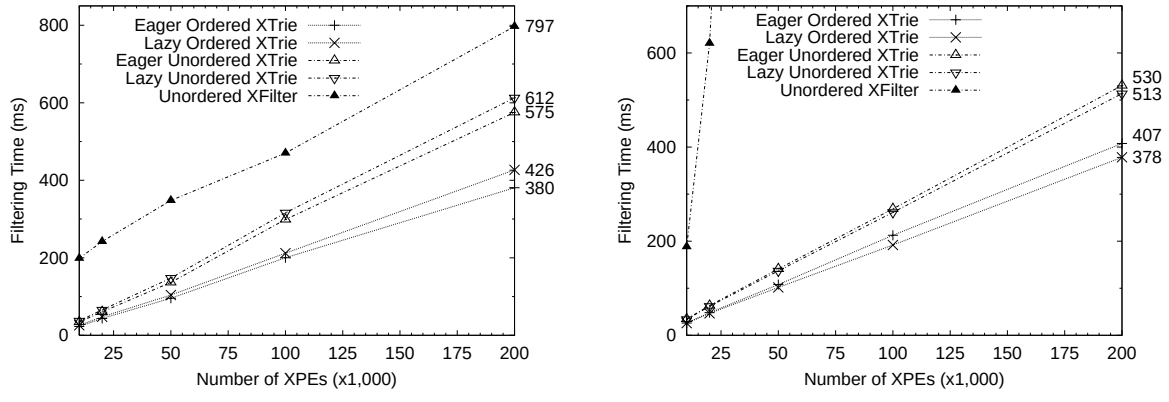
[6] We could not represent this behavior in Fig. 12b as it would have required scaling down the vertical axis by two orders of magnitude.

[7] The data points of Unordered XFilter for tree-structured XPEs do not appear on the Fig. 14b due to the scale and the extremely slow performance of the algorithm.



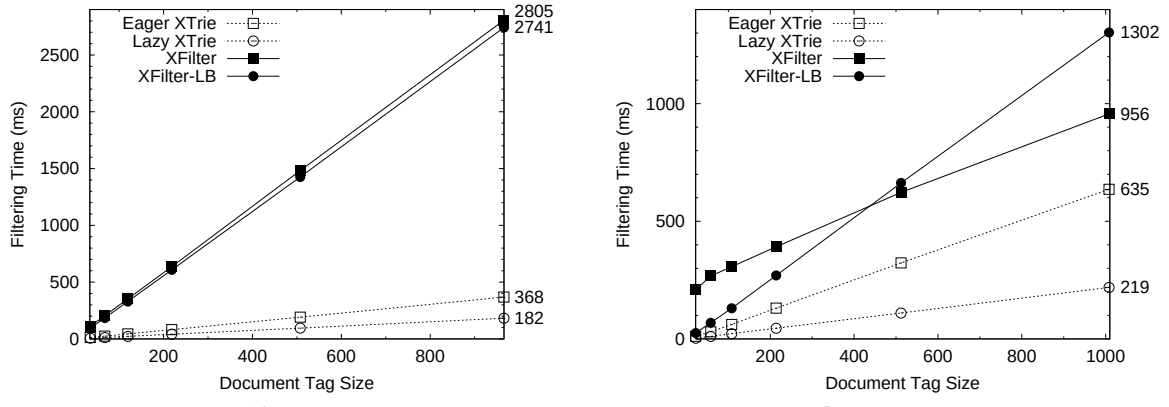
a 10 DTDs **b** NITF DTD

Fig. 11. Varying P for Single-path XPEs ($L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0, \theta = 0, T \approx 100$)



a 10 DTDs **b** NITF DTD

Fig. 12. Varying P for tree-structured XPEs ($L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0.1, \theta = 0, T \approx 100$)

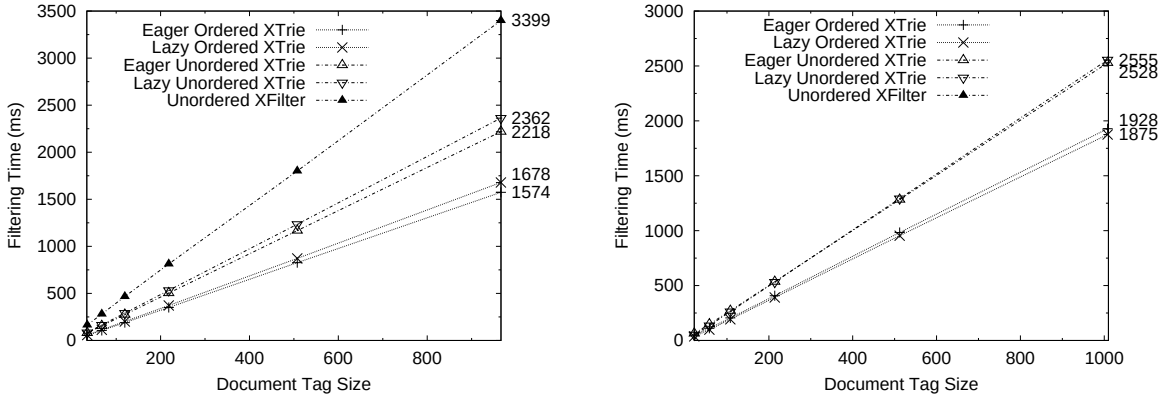


a 10 DTDs **b** NITF DTD

Fig. 13. Varying document length T for single-path XPEs ($P = 100K, L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0, \theta = 0$)

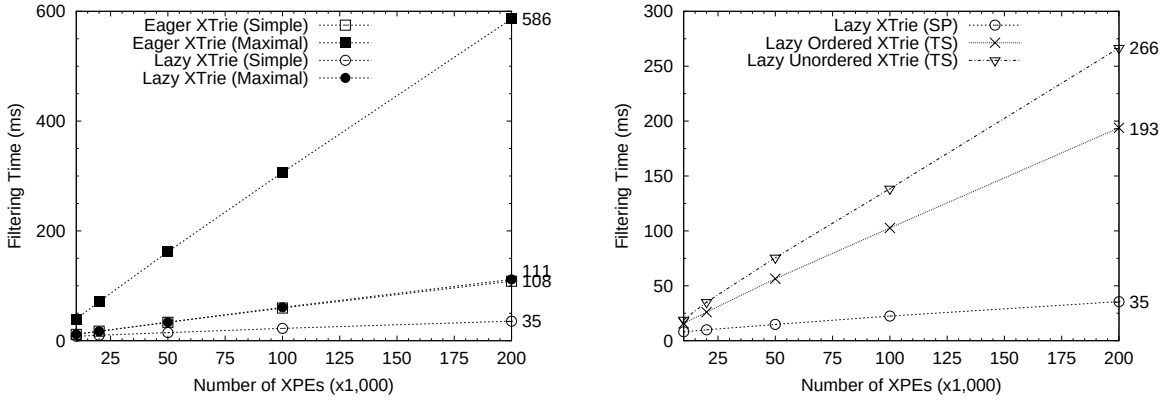
7.2.2 Effect of decomposition

As previously mentioned, XTrie is expected to perform better as the length of substrings grows, because the substring table is accessed only when an entire substring has been matched. To measure the gain from matching substrings instead of single elements, we have compared the performance of XTrie with two different decompositions: the *simple* decomposition (indicated as Simple) introduced in Sect. 3 that generates a small number of long substrings, and a simplistic decomposition called *maximal* decomposition (indicated as Maximal) that basically generates a substring for each individual element (hence giving rise to a maximal number of substrings). As shown in Fig. 15a, XTrie performs significantly better with the simple decomposition. The improvement is more noticeable for Eager XTrie than for Lazy XTrie, because the former algorithm accesses the substring table for each matching substring (i.e., for each element when using the maximal decomposition) while the latter only accesses the substring table when encountering a leaf substring; therefore, the gain from using



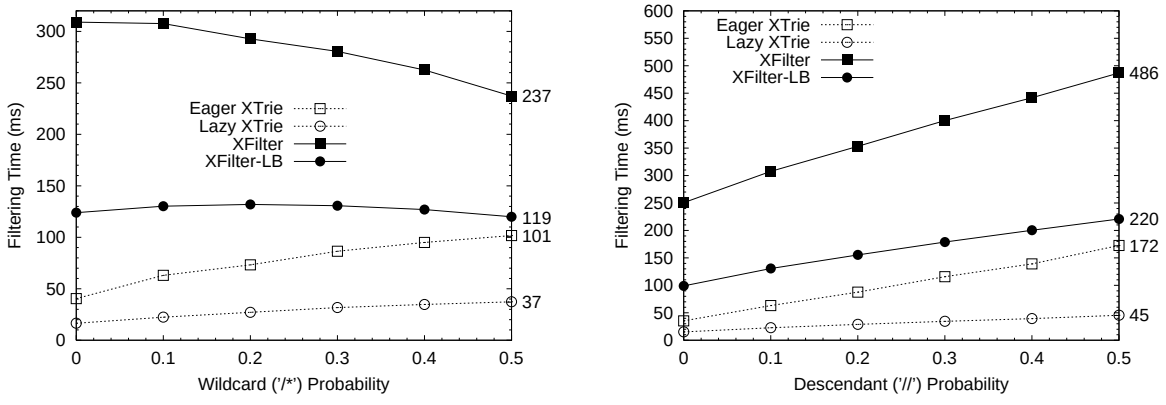
a 10 DTDs **b** NITF DTD

Fig. 14. Varying document length T for tree-structured XPEs ($P = 50K, L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0.1, \theta = 0$)



a Decomposition **b** Algorithm Specialization

Fig. 15. Effect of decomposition and algorithm specialization for single-Path XPEs with NITF DTD ($L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0, \theta = 0, T \approx 100$)



a Effect of p_* ($p_{//} = 0.1$) **b** Effect of $p_{//}$ ($p_* = 0.1$)

Fig. 16. Varying p_* and $p_{//}$ with NITF DTD ($P = 100,000, L = 20, p_\lambda = 0, \theta = 0, T \approx 100$)

longer substrings is proportionally larger for Eager XTrie than for Lazy XTrie.

7.2.3 Effect of algorithm specialization

We have presented different variants of the XTrie filtering algorithm, adapted to different types of XPEs (single-path, tree-structured ordered and unordered). While the tree-structured variants can be used to filter single-path XPEs, the variants optimized for single-path XPEs are expected to be more efficient. We have measured the gain of using specialized algorithms (indicated with SP for single-path algorithms) instead of their generic counterparts (indicated with TS for tree-structured algorithms) for filtering single-path XPEs. The results, shown in Fig. 15b (for clarity, we only represented the results for the lazy algorithms as the eager ones had a similar behavior), clearly demonstrate the benefits of using the optimized

algorithms. The tree-structured algorithms suffer from more complex data structures and matching procedures, which degrade their performance.

7.2.4 Effect of wildcards and descendant operators

Figures 16a and b show the influence of wildcard and descendant operators on filtering speed. Interestingly, XFilter performs slightly better as the number of wildcards increases, while the performance of XTrie degrades. This can be explained by the fact that XFilter does not use nodes for representing wildcards; thus, the number of nodes decreases when the percentage of wildcards increases. The performance of XTrie degrades because substrings become shorter when the number of wildcard increases; as previously mentioned, XTrie degenerates to a hash table when substrings become single elements. Both XFilter and XTrie are affected by the probability of descendant operators, but the performance of XTrie degrades slightly more than XFilter. As with wildcards, this can be explained by the shorter length of the substrings. Lazy XTrie is less affected by higher probabilities of wildcard and descendant operators than Eager XTrie, because Lazy XTrie only incurs the cost of accessing the substring table when reaching a leaf node, and the number of leaf nodes remains constant as the number of substring grows. We observed the same behavior with tree-structured XPEs and (un)ordered XTrie.

7.2.5 Effect of XPE depth

We have measured the influence of the XPEs' average depth on the performance of the filtering algorithms. As show in Fig. 17a, the performance of all algorithms degrades as the number of levels grow. The basic variant of XFilter is the most affected, because longer substrings result is a larger number of elements and yields longer candidate lists. The list balancing optimization attenuates this problem. Similarly, the performance of XTrie degrades because the number of substrings increases with the XPEs' depth.

7.2.6 Effect of skew

Figure 17b shows the behavior of the filtering algorithms when element names are skewed (note that the XML documents are not skewed). The graph shows that the performance of XFilter (without the list balancing optimization) increases sharply with highly skewed XPEs. This is due to the fact only a small number of candidate lists become large – those associated with skewed elements. All other lists remain small. As XML documents are not skewed, all lists are accessed approximately the same number of times, and small lists improve the algorithm's performance. With the list balancing optimization, XFilter strives to keep candidate lists small and is less affected by skewed data. As expected, XTrie is also less sensitive to skew because it indexes substrings rather than element names, and the number of identical substrings remains significantly smaller than the number of identical element name with high skew. The performance of Lazy XTrie even improves significantly with highly skewed XPEs. As explained before, the lazy algorithm only accesses the substring table upon matching of a leaf substring, and since the XML data is not skewed, this happens less frequently with higher skew.

7.3 Space analysis

Since XTrie essentially uses bounded data structures at runtime, its space requirement depends only on the set of XPEs. In contrast, XFilter makes extensive use of dynamic lists for storing candidate path nodes and its space requirements depends on both the XPEs and the input XML documents. Figures 18a and b compare the memory usage of the various algorithms[8] as a function of the number of XPEs for single-path XPEs and tree-structured XPEs, respectively. It appears clearly that XTrie has lower memory requirements than XFilter, and that the algorithms specialized for single-path XPEs use less resources. Interestingly, the faster variants (Lazy XTrie and XFilter-LB) are also those that use less memory. Although the space requirements of XFilter can be asymptotically very large, in practice they remain reasonable and none of the algorithms had problems maintaining their index structures in main memory.

8 Conclusions

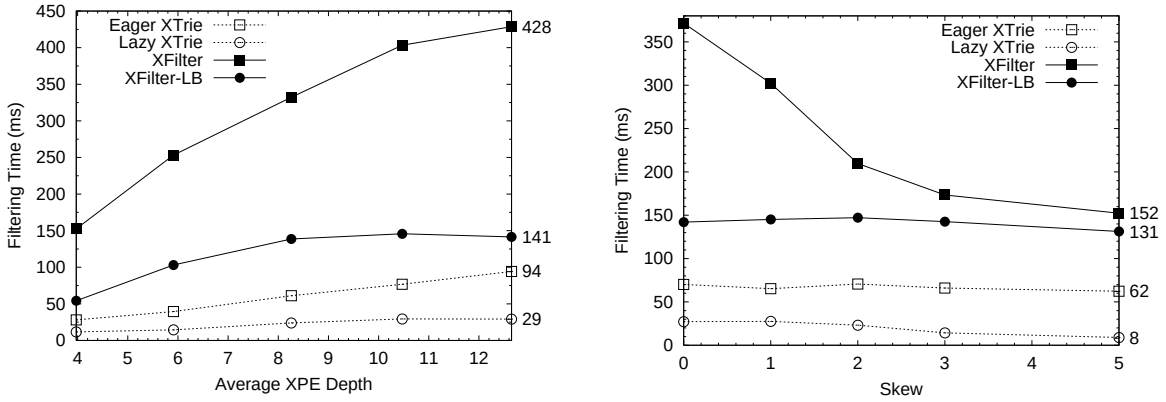
In this paper, we have proposed a novel index structure, termed XTrie, that supports the efficient filtering of streaming XML documents based on XPath expressions. Our XTrie index offers several novel features that make it especially attractive for large-scale publish/subscribe systems. First, the XTrie 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 the anonymous reviewers for their detailed comments which helped significantly to improve the presentation of this paper.

A Maintenance algorithms for XTrie

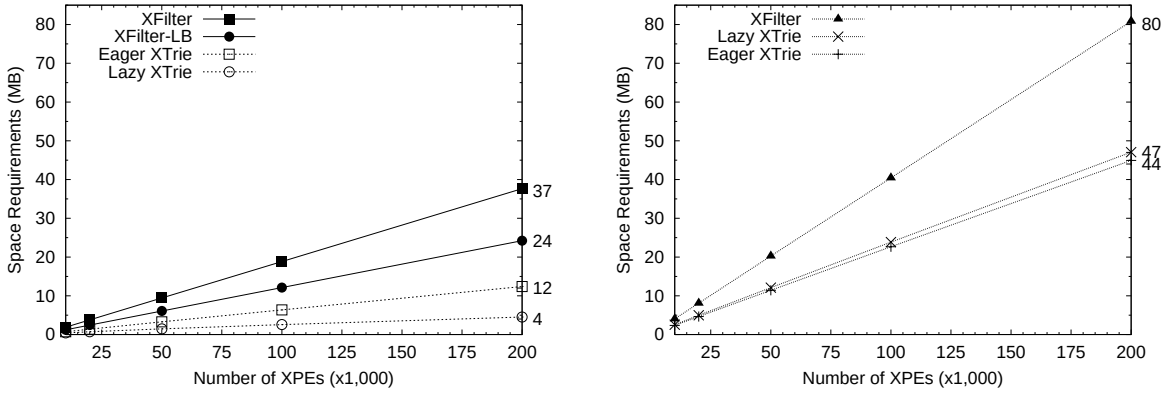
In this section, we present the maintenance algorithms for XTrie. The maintenance of XTrie is overall rather straightforward except for the maintenance of the max-suffix pointers in the trie T which is more involved. One approach to efficiently

[8] Ordered and unordered XTrie have the same memory usage, and are not represented separately in the figures.



a Effect of L ($\theta = 0$) **b** Effect of θ ($L = 20$)

Fig. 17. Varying L and θ with NITF DTD ($P = 100,000, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0, T \approx 100$)



a Single-Path XPEs **b** Tree-Structured XPEs

Fig. 18. Memory usage with NITF DTD ($L = 20, p_* = 0.1, p_{//} = 0.1, p_\lambda = 0, \theta = 0, T \approx 100$)

maintain these pointers is to build an auxiliary trie structure, denoted by T_{rev}, on the set of reversed substrings, which we now describe.

The auxiliary structure T_{rev} is basically a *suffix trie* on the set of reverse substrings[9] in $\mathcal{S}$; i.e., T_{rev} is a trie on the set $\{s' \mid s \in \mathcal{S},\ s' \text{ is a suffix of } reverse(s)\}$[10]. The nodes in T and T_{rev} are related by the following invariant condition: for each node node N in T, there exists a unique node N' in T_{rev} such that $label(N) = reverse(label(N'))$. We explicitly maintain this association between the nodes in T and T_{rev} by enhancing T_{rev} with additional pointers as follows: for each node N' in T_{rev}, we maintain a special pointer, denoted by $\gamma(N')$, that points to the node N in T (if it exists in T) such that $label(N) = reverse(label(N'))$; otherwise, $\gamma(N')$ is initialized to a null pointer value. Note that for the root node N'_{root} of T', $\gamma(N'_{root})$ points to the root node of T. Given T_{rev}, the max-suffix pointer value of a node N in T, $\beta(N)$, can be easily computed as follows. Let $\mathcal{P} =< N'_1, N'_2, \cdots, N'_k >$ denote the unique path of nodes in T_{rev} beginning from the root node N'_1 down to some node N'_k such that $label(N'_k) = reverse(label(N))$. Then, the value of $\beta(N)$ is given by $\gamma(N'_i)$, where N'_i ($1 \leq i < k$) is the bottom-most node in $\mathcal{P}$ (excluding node N'_k) that has a non-null pointer value. Note that such a node always exists since $\gamma(N'_1)$ points to the root node of T.

[9] The reverse of a substring s, denoted by $reverse(s)$, is defined to be $e_n.e_{n-1}.\cdots.e_1$ if $s = e_1.e_2.\cdots.e_n$ is a concatenation of n element names.

[10] Note that since the set of suffixes of s' is a proper subset of the set of suffixes of s if s' is a proper prefix of s, it is sufficient to construct the suffix trie T_{rev} on the reverse substrings in $\{label(N) \mid \text{N is a leaf node in T}\}$.

A.1 Maintenance algorithm for new insertions

This section presents an algorithm (shown in Fig. 19) to update $XTrie$ and the auxiliary structure T_{rev} when a new set of XPEs $\mathcal{P}_{new}$ is to be added. The maintenance algorithm to handle insertion of new XPEs consists of three main phases. The first phase expands T with new nodes (if required) and updates ST with a new entry for each substring in the simple decomposition of each new XPE. The second phase expands T_{rev} with new nodes if new nodes have been inserted into T during the first phase, and updates their $\gamma()$ pointer values; this phase also updates the max-suffix pointers of some of the existing nodes in T. Finally, the third phase updates the max-suffix pointers of the new nodes that were added to T in the first phase. We now elaborate on the details of these three phases.

In the first phase (Steps 1 to 11), for each substring s in the simple decomposition of each new XPE, we traverse T using s to first check if there exists a node N in T such that

Algorithm INSERT-XPE ($\mathcal{P}_{new}$, ST, T)
Input: $\mathcal{P}_{new}$ is a set of XPEs to be inserted; (ST, T) is an XTrie index.
Output: An updated XTrie.
1) Initialize $Node_{new}$ to be empty;
2) **for each** $p \in \mathcal{P}_{new}$ **do**
3) **for each** substring s in the simple decomposition of p **do**
4) Traverse T using s and let N be the last node visited in T;
5) **if** ($label(N) \neq s$) **then**
6) Append a new path of nodes $< N_1, N_2, \cdots, N_k >$ to N such that $label(N_k) = s$;
7) Initialize $\alpha(N_i) = null$ for $i = 1, 2, \ldots, k-1$;
8) Initialize $\alpha(N_k)$ to point to a new entry in ST;
9) Initialize $\beta(N_i)$ to point to the root node of T for $i = 1, 2, \ldots, k$;
10) $Node_{new} = Node_{new} \cup \{N_1, N_2, \cdots, N_k\}$;
11) Insert a new entry into ST for s;
12) **for each** $N \in Node_{new}$ **do**
13) Traverse T_{rev} using $reverse(label(N))$ and let N' be the last node visited in T_{rev};
14) **if** ($label(N') \neq reverse(label(N))$) **then**
15) Append a new path of nodes $< N'_1, N'_2, \cdots, N'_j >$ to N' such that $label(N'_j) = reverse(label(N))$;
16) Initialize $\gamma(N'_i)$ to point to the root node of T for $i = 1, 2, \ldots, j-1$;
17) Initialize $\gamma(N'_j)$ to point to node N;
18) **else**
19) Update $\gamma(N')$ to point to N;
20) Let $Node_{\max} = \{N'' \text{ is a descendant node of } N' \mid \gamma(N'') \neq null, \gamma(X) = null \text{ for each node } X \text{ between } N' \text{ and } N''\}$;
21) **for each** node $N'' \in Node_{\max}$ **do**
22) Let N_t be the node in T pointed to by $\gamma(N'')$;
23) Update $\beta(N_t)$ to point to N;
24) **for each** $N \in Node_{new}$ **do**
25) Traverse T_{rev} using $reverse(label(N))$ and let $P =< N'_1, N'_2, \cdots N'_m >$ denote the path of nodes visited;
26) Update $\beta(N)$ to $\gamma(N'_j)$, where j is the maximum value in $[1, m-1]$ such that $\gamma(N'_j) \neq null$;

Fig. 19. Algorithm to maintain XTrie for insertion of a set of XPEs

$label(N) = s$. If not, an appropriate path of new nodes is inserted into T so that a leaf node in T is reachable using s. A new entry corresponding to s is also inserted into the substring-table ST. The $\alpha(.)$ pointer values are updated appropriately, while the $\beta(.)$ are simply initialized to point to the root node of T at this point.

In the second phase (Steps 12 to 23), we update T_{rev} to maintain the invariant condition for the newly inserted nodes in T. Therefore, for each newly inserted node N in T, we traverse T_{rev} using $reverse(label(N))$ to check if there exists a node N' in T_{rev} such that $label(N) = reverse(label(N'))$. There are two possible cases. In the first case, if N' does not exist in T_{rev}, then an appropriate path of new nodes (with leaf node N') is inserted into T_{rev} so that $label(N) = reverse(label(N'))$. The $\gamma(.)$ pointer values for the newly inserted nodes in T_{rev} are updated appropriately. In the second case, if N' already exists in T_{rev}, then it is necessary that $\gamma(N')$ has a null pointer value (otherwise, it would imply that node N already exists in T contradicting the fact that N is a newly inserted node in T); thus, what remains to be done is to simply update $\gamma(N')$ to point to N. Since there might be some existing nodes in T whose max-suffix pointer values were initialized to $\gamma(N')$, we therefore need to update the max-suffix pointer values of such nodes to point to N instead. This update is performed in Steps 20 to 23. The set of nodes in T_{rev} associated with the affected nodes in T are represented by the set $Node_{\max}$; i.e., for each node $N'' \in Node_{\max}$, we need to update the max-suffix pointer of the node pointed to by $\gamma(N'')$. Note that the number of such affected nodes is bounded by the branching degree of node N' in T_{rev}. Finally, the third phase (Steps 24 to 26), updates the max-suffix pointers for the newly inserted nodes in T as described in the previous section.

A.2 Maintenance algorithm for deletions

This section presents an algorithm (shown in Fig. 20) to update $XTrie$ when an existing XPE p is to be deleted. The maintenance algorithm consists of two main phases. The first phase deletes the appropriate entries in the ST that correspond to the substrings in the simple decomposition of p; nodes in T that have become "useless" as a result of the changes in ST are also deleted. The second phase deletes nodes in T_{rev} that have become "useless" and also updates those max-suffix pointers in T that are now pointing to non-existing nodes as a consequence of the nodes deleted in the first phase.

In the first phase (Steps 1 to 11), for each substring s in the simple decomposition of p, we delete the corresponding entry to s in ST by navigating to T via T_{rev}; that is we first traverse T_{rev} using $reverse(s)$ to reach a node N' in T_{rev} and

Algorithm DELETE-XPE (p, ST, T)
Input: p is a XPE to be deleted; (ST, T) is an XTrie index.
Output: An updated XTrie.
1) Let S_p be the set of distinct substrings from the simple decomposition of p;
2) Let S_{sort} be the sorted sequence of substrings in S_p in descending order of the substring length;
3) **for each** substring s in S_{sort} **do**
4) Traverse T_{rev} using $reverse(s)$ to reach node N';
5) Let N be the node in T pointed to by $\gamma(N')$;
6) Access ST using $\alpha(N)$ to delete the entry corresponding to s;
7) **if** (the deleted entry is the last entry for s) **then**
8) Update $\alpha(N)$ to the null pointer value;
9) **if** (N is a leaf node) **then**
10) Delete N from T;
11) Mark the node N' ;
12) **for each** substring s in S_{sort} **do**
13) Traverse T_{rev} using $reverse(s)$ to reach node N';
14) **if** (N' is marked) **then**
15) Let N_{anc} be the closest ancestor node of N' such that N_{anc} is not marked and $\gamma(N_{anc}) \neq null$;
16) Let $Node_{\max} = \{N''$ is a descendant node of $N' \mid \gamma(N'') \neq null$, N" is not marked, for each node X between N_{anc} and N'', X is marked or $\gamma(X) = null\}$;
17) **for each** node $N'' \in Node_{\max}$ **do**
18) Let N be the node in T pointed to by $\gamma(N'')$;
19) Update $\beta(N)$ to point to $\gamma(N_{anc})$;
20) **if** (N' is a leaf node) **then**
21) Delete N' from T_{rev};
22) **else**
23) Update $\gamma(N')$ to a null pointer value;

Fig. 20. Algorithm to maintain XTrie for an XPE deletion

then navigate to its associated node N in T using $\gamma(N')$. The reason for this indirect navigation is because we need to "take note" of node N' in T_{rev} (by marking that node) if the node N in T is deleted. Note that a node N in T will be deleted if has become "useless"; i.e., N has become a leaf node and the value of $\alpha(N)$ has become a null pointer value. In order to efficiently ensure that all the useless nodes in T are deleted, we need to visit these to-be-deleted nodes in a bottom-up manner; otherwise, we would have missed deleting an internal node that later becomes a useless leaf node. For this reason, we iterate through the to-be-deleted substrings in descending order of their lengths by first sorting them into the sequence S_{sort}. For each node in T that is deleted, its associated node in T_{rev} is marked for further processing in the second phase.

The second phase (Steps 12 to 23) begins once all the relevant entries in ST and useless nodes in T have been deleted. The purpose of this phase is to delete useless nodes in T_{rev} and update the max-suffix pointers in T using the updated T_{rev}. For each deleted node N in T, we first navigate to its associated marked node N' in T_{rev}. Node N' is deleted from T_{rev} if N' is a leaf node; otherwise, we update $\gamma(N')$ to a null pointer value. The updating of the affected max-suffix pointers in T, which is performed in Steps 15 to 19, is similar to the procedure described earlier for the second phase in Algorithm INSERT-XPE.

B Search algorithm for lazy XTrie

In this section, we present the detailed search algorithm for Lazy XTrie. The main algorithm is shown in Fig. 21; a comparison between the lazy and eager variants was described in Sect. 5.1.

Algorithm LAZY-MATCH-SUBSTRING (shown in Fig. 22) is called to iterate through each instance of s in the indexed substrings via the linked list associated with s when a matching leaf substring s is detected; the input parameter r refers to the first row in the substring-table that corresponds to s. For each matching substring $s_{i,j} \in \mathcal{S}_{leaf}$ (matching at level ℓ and corresponding to row r in ST), Algorithm MATCH-SUBSTRING-SUB (shown in Fig. 23) is invoked to check if this matching is a partial matching of $s_{i,j}$ and, if so, whether it also completes the matching of p_i. The algorithm returns one of the following three status values: $completeMatch$ if there is a matching of p_i, $partialMatch$ if there is a partial matching of $s_{i,j}$ at level ℓ, or $noMatch$ otherwise. The input parameter $subpatternMatch$ is a Boolean variable indicating whether or not there is a matching of the subpattern[11] rooted at $s_{i,j}$ (with $s_{i,j}$ matching at level ℓ); and the input parameter $childSubpatternMatch$ is a Boolean variable indicating whether or not there is a matching of the subpattern rooted at the most recently detected child substring of $s_{i,j}$. For the non-trivial case where $s_{i,j}$ is a non-root

[11] The *subpattern* rooted at a substring s of a XPE p refers to a XPE p' is derived from p that consists of only all the substrings in the subtree rooted at s in p.

Algorithm LAZY-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) Let $\mathcal{C}$ be a $|ST| \times L_{\max}$ bit-array with all values initialized to 0;
5) Let $\mathcal{M}$ be a $|\mathcal{S}| \times L_{\max}$ bit-array with all values initialized to 0;
6) Initialize $\ell = 0$; // ℓ is the current document level
7) Initialize N to be the root node of T; // N is the current trie node
8) **repeat**
9) **if** (a start-tag t is parsed in D) **then**
10) $\ell = \ell + 1$;
11) **while** ((there is no edge labelled "t" from N) **and** (N is not the root node of T)) **do**
12) $N = \beta(N)$;
13) **if** (there is an edge labelled "t" from N to N' in T) **then**
14) $Node[\ell] = N'$; $N = N'$;
15) **while** (N' is not the root node) **do**
16) $r = ABS(\alpha(N'))$; // r is the absolute value of $\alpha(N')$
17) **if** ($r \neq 0$) **then**
18) Set $\mathcal{M}[ST[r].SID, \ell]$ to 1;
19) **if** ($\alpha(N') > 0$) **then**
20) $R = R \cup$ LAZY-MATCH-SUBSTRING ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, \alpha(N'), \ell$);
21) $N' = \beta(N')$;
22) **else if** (an end-tag is parsed in D) **then**
23) Reset $\mathcal{B}[i, \ell]$ to 0 for i = 1 to $|ST|$;
24) Reset $\mathcal{M}[i, \ell]$ to 0 for i = 1 to $|\mathcal{S}|$;
25) $Node[\ell]$ = root node of T;
26) $\ell = \ell - 1$;
27) Reset $\mathcal{C}[i, \ell]$ to 0 for i = 1 to $|ST|$;
28) $N = Node[\ell]$;
29) **until** (D has been completely parsed);
30) **return** R;

Fig. 21. Algorithm to search lazy XTrie

Algorithm LAZY-MATCH-SUBSTRING (ST, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{M}$, r, ℓ)
Input: ST is the substring-table of an XTrie index. $\mathcal{B}$ is a 2-dimensional integer-array.
$\mathcal{C}$ is a 2-dimensional bit-array.
$\mathcal{M}$ is a 2-dimensional bit-array.
r refers to the first row in ST that corresponds to some leaf substring that matches at level ℓ.
Output: Set of matching XPEs.
1) Initialize R to be empty;
2) **while** ($r \neq 0$) **do**
3) $status$ = MATCH-SUBSTRING-SUB ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r, \ell, true, true$);
4) **if** ($status == completeMatch$) **then**
5) Insert the id. of the XPE corresponding to row r into R;
6) $r = ST[r].Next$;
7) **return** R;

Fig. 22. Algorithm to process a matching substring in lazy XTrie

substring, the algorithm checks if the matching of $s_{i,j}$ at level ℓ is a partial matching by iterating through each possible level ℓ' for which the parent substring of $s_{i,j}$ (corresponding to row r' in ST) can be matched (i.e., $\ell - \ell' \in ST[r].RelLevel$) in Steps 13 to 27. There are three possible cases to consider. In the first case, if $\mathcal{B}[r', \ell'] > ST[r].Rank$, then the matching is a redundant matching of $s_{i,j}$ and it can be ignored. In the second case, if $\mathcal{B}[r', \ell'] = ST[r].Rank$, then the matching is a non-redundant matching of $s_{i,j}$; in addition, if the matching is also a subtree-matching of $s_{i,j}$ (Step 16), then Algorithm PROPAGATE-UPDATE (in Fig. 9) is invoked to check if this leads to subtree-matchings of the ancestor substrings of $s_{i,j}$ and possibly a complete matching of p_i. In the third and final case, where $\mathcal{B}[r', \ell'] < ST[r].Rank$, we have two possible sub-cases to consider. If $\mathcal{B}[r', \ell'] > 0$, then there exists at least one preceding sibling substring of $s_{i,j}$ that has not been matched yet, which implies that the matching of $s_{i,j}$ is not a partial matching and can therefore be ignored. Otherwise, if $\mathcal{B}[r', \ell'] = 0$, then in order for the matching of $s_{i,j}$ to be a partial matching, it is necessary that there is a partial matching of the parent substring of $s_{i,j}$ at level ℓ' and $s_{i,j}$ is its first child substring. Therefore, a recursive call to

Algorithm MATCH-SUBSTRING-SUB
(ST, $\mathcal{B}$, $\mathcal{C}$, $\mathcal{M}$, r, ℓ, $subpatternMatch$, $childSubpatternMatch$)
Output: Returns one of the following status values:
(1) $completeMatch$ if there is a matching of p_i,
(2) $partialMatch$ if there is a partial matching of $s_{i,j}$ at level ℓ, or
(3) $noMatch$, otherwise.

1) Initialize $status = noMatch$;
2) $r' = ST[r].ParentRow$;
3) **if** ($r' == 0$) **then** //r corresponds to a root substring
4) **if** ($\ell \in ST[r].RelLevel$) **then**
5) $status = partialMatch$;
6) **else** //r corresponds to a non-root substring
7) **if** ($subpatternMatch$ **and** ($ST[r'].NumChild == ST[r].Rank$)) **then**
8) $parentSubpatternMatch = true$;
9) **else**
10) $parentSubpatternMatch = false$;
11) $parentSid = ST[r'].SID$;
12) Initialize $\ell' = \ell - \ell_{\min}$, where $ST[r].RelLevel = [\ell_{\min}, \ell_{\max}]$;
13) **while** ($status \neq completeMatch$) **and** ($\ell' > 0$) **and** ($\ell - \ell' \in ST[r].RelLevel$) **do**
14) **if** ($\mathcal{B}[r', \ell'] == ST[r].Rank$) **and** ($\mathcal{C}[r', \ell'] == 0$) **then**
15) $status = partialMatch$;
16) **if** ($parentSubpatternMatch$) **then**
17) $\mathcal{B}[r', \ell'] = ST[r'].NumChild + 1$;
18) **if** ($ST[r'].ParentRow == 0$) **or** (PROPAGATE-UPDATE ($ST, \mathcal{B}, \mathcal{C}, r', \ell'$)) **then**
19) $status = completeMatch$;
20) **else if** ($subpatternMatch$) **then**
21) $\mathcal{B}[r', \ell'] = \mathcal{B}[r', \ell'] + 1$;
22) $\mathcal{C}[r', \ell'] = 1$;
23) **else if** ($\mathcal{M}[parentSid, \ell']$ **and** ($\mathcal{B}[r', \ell'] == 0$) **and** ($ST[r].Rank == 1$)) **then**
24) ret = MATCH-SUBSTRING-SUB ($ST, \mathcal{B}, \mathcal{C}, \mathcal{M}, r', \ell', parentSubpatternMatch, subpatternMatch$);
25) **if** ($ret \neq noMatch$) **then**
26) $status = ret$;
27) $\ell' = \ell' - 1$;
28) **if** ($status == partialMatch$) **then**
29) **if** ($subpatternMatch$) **then**
30) $\mathcal{B}[r, \ell] = ST[r].NumChild + 1$;
31) **if** ($r' == 0$) **then**
32) $status = completeMatch$;
33) **else**
34) **if** ($\mathcal{B}[r, \ell] == 0$) **then**
35) $\mathcal{B}[r, \ell] = 1$;
36) **if** ($childSubpatternMatch$) **then**
37) $\mathcal{B}[r, \ell] = \mathcal{B}[r, \ell] + 1$;
38) $\mathcal{C}[r, \ell] = 1$;
39) **return** $status$;

Fig. 23. Auxiliary algorithm to process a matching substring in lazy XTrie

Algorithm MATCH-SUBSTRING-SUB is made in Step 24 to check if there is a partial matching of its parent substring at level ℓ'. Depending on the status of the matching of $s_{i,j}$, its $\mathcal{B}$ entry is updated accordingly in Steps 28 to 38.

C Eager XTrie for single-path XPEs

We discuss how Eager XTrie can be optimized for the special case where the indexed XPEs are all single-path XPEs. Basically, Eager XTrie for single-path XPEs differs from Eager XTrie for tree-structured XPEs in the following ways. First, since each substring in a single-path XPE has at most one child substring, the substring-table ST for single-path XPEs is simpler than that for tree-structured XPEs; specifically, each row in ST (corresponding to some substring $s_{i,j}$) is a 3-tuple ($RelLevel$, $RootSubstr$, $Next$), where $RelLevel$ and $Next$ are defined equivalently as before; and $RootSubstr$ is a single bit that is set to 1 if and only if $s_{i,j}$ is the root substring of p_i (i.e., $j = 1$). Similar to Lazy XTrie for single-path XPEs, the attributes $Rank$, $NumChild$, and $ParentRow$, which are necessary for tree-structured XPEs, are not needed for single-path XPEs. Furthermore, it is sufficient for the run-time information $\mathcal{B}$ of Eager XTrie to be a bit-array (rather than an

Algorithm MATCH-SUBSTRING($ST, \mathcal{B}, r, \ell$)
Input: ST is the substring-table of an Eager XTrie index.
$\mathcal{B}$ is a 2-dimensional bit-array. r refers to the first row in ST that corresponds to some substring that matches at level ℓ.
Output: Set of matching XPEs.
1) Initialize R to be empty;
2) **while** ($r \neq 0$) **do**
3) $match = false$;
4) **if** ($ST[r].RootSubstr == 1$) **then**
5) **if** ($\ell \in ST[r].RelLevel$) **then**
6) $match = true$;
7) **else if** ($\exists\ \ell' \in [1, \ell - 1]$ such that $\ell - \ell' \in ST[r].RelLevel$ and $\mathcal{B}[r - 1, \ell'] == 1$) **then**
8) $match = true$;
9) **if** ($match$) **then**
10) Set $\mathcal{B}[r, \ell]$ to 1;
11) **if** (($r == |ST|$) **or** ($ST[r + 1].RootSubstr == 1$)) **then**
12) Insert the id. of the XPE corresponding to row r into R;
13) $r = ST[r].Next$;
14) **return** R;

Fig. 24. Algorithm to process a matching substring in eager XTrie (for single-path XPEs)

integer-array) such that $\mathcal{B}[r_{i,j}, \ell] = 1$ if and only if there is a partial matching of $s_{i,j}$ at level ℓ.

The main search algorithm for single-path XPEs is similar to that for tree-structured XPEs (in Fig. 7) except that $\mathcal{B}$ is now a bit-array. The matching algorithm is, however, simpler for single-path XPEs and is shown in Fig. 24.

References

1. M.K. Aguilera, R.E. Strom, D.C. Sturman, M. Astley, T. D. Chandra (1999) Matching events in a content-based subscription system. In: Proc. ACM PODC, pp 53–61, Atlanta, Ga., USA
2. M. Altinel, M.J. Franklin (2000) Efficient filtering of XML documents for selective dissemination of information. In: Proc. VLDB, pp 53–64
3. S. Amer-Yahia, S. Cho, L.V.S. Lakshmanan, D. Srivastava (2001) Minimization of tree pattern queries. In: Proc. ACM SIGMOD, pp 497–508, Santa Barbara, Calif., USA
4. S. Amer-Yahia, S. Cho, D. Srivastava (2002) Tree pattern relaxation. In: Proc. EDBT, pp 496–513, Prague, Czech Republic
5. Apache (2001) Xerces C++ parser. http://xml.apache.org
6. A. Carzaniga, D.S. Rosenblum, A.L. Wolf (2001) Design and evaluation of a wide-area event notification service. ACM Trans Comput Syst 19(3):332–383
7. The Intel Corporation (2000) Intel netStructure XML accelerators. http://www.intel.com/netstructure/products/xml_accelerators.htm
8. R. Cover (1999) The SGML/XML web page. http://www.oasis.open.org/cover/sgml-xml.html
9. A.L. Diaz, D. Lovell (1999) XML Generator. http://www.alphaworks.ibm.com/tech/xmlgenerator
10. E.N. Hanson, M. Chaabouni, C.-H. Kim, Y.-W. Wang (1990) A predicate matching algorithm for database rule systems. In: Proc. ACM SIGMOD, pp 271–280, Atlantic City, N.J., USA
11. F. Fabret, H.A. Jacobsen, F. Llirbat, K.A. Ross, D. Shasha (2001) Filtering algorithms and implementations for very fast publish/subscribe systems. In: Proc. ACM SIGMOD, pp 115–126, Santa Barbara, Calif., USA
12. E.N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J.B. Park, A. Vernon (1999) Scalable trigger processing. In: Proc. IEEE ICDE, pp 266–275, Sydney, Australia
13. D.E. Knuth (1998) The art of computer programming: sorting and searching, vol. 3, 2nd edn. Addison-Wesley, Reading, Mass., USA
14. D. Megginson (2002) SAX: a simple API for XML. http://www.megginson.com/SAX/
15. B. Nguyen, S. Abiteboul, G. Cobena, M. Preda (2001) Monitoring XML data on the Web. In: Proc. ACM SIGMOD, pp 437–448, Santa Barbara, Calif., USA
16. P. Ramanan (2002) Efficient algorithms for minimizing tree pattern queries. In: Proc. ACM SIGMOD, pp 299–309, Madison, Wis., USA
17. T. Schlieder (2002) Schema-driven evaluation of approximate tree-pattern queries. In: Proc. EDBT, pp 514–532, Prague, Czech Republic
18. B. Segall, D. Arnold, J. Boot, M. Henderson, T. Phelps (2000) Content-based routing with Elvin4. In: AUUG2K, Canberra, Australia
19. W3C (2002) Document object model (DOM) level 1 specification (2nd edn), Version 1.0. http://www.w3.org/TR/REC-DOM-Level-1/
20. W3C (1999) XML path language (XPath) 1.0. http://www.w3.org/TR/xpath
21. W3C (2000) Extensible markup language (XML) 1.0, 2nd edn. http://www.w3.org/TR/REC-xml/
22. G.K. Zipf (1949) Human behaviour and principle of least effort. Addison-Wesley, Cambridge, Mass., USA