GPX-Matcher: A Generic Boolean Predicate-based XPath Expression Matcher

Mohammad Sadoghi, Ioana Burcea, and Hans-Arno Jacobsen
Middleware Systems Research Group
Department of Electrical and Computer Engineering
Department of Computer Science
University of Toronto, Canada
mo@cs.toronto.edu, ioana@eecg.toronto.edu, jacobsen@eecg.toronto.edu

ABSTRACT

Content-based architectures for XML data dissemination are gaining increasing attention both in academia and industry. These dissemination networks are the building blocks of selective information dissemination applications which have wide applicability such as sharing and integrating information in both scientific and corporate domains. At the heart of these dissemination services is a fast engine for matching of an incoming XML message against stored XPath expressions to determine interested consumers for the message. To achieve the ultra-low response time, predominant in financial message processing, the XPath expression matching must be done efficiently. In this paper, we develop and evaluate a novel algorithm based on a unique encoding of XPath expressions and XML messages, unlike dominating automaton-based algorithms, for efficiently solving this matching problem. We demonstrate a matching time in the millisecond range for millions of XPath expressions which significantly outperforms state-of-the-art algorithms.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Design, Measurement, Experimentation

Keywords

Publish/Subscribe, Event Processing, Complex Event Processing, Matching Problem and Algorithm, and ToPSS

1. INTRODUCTION

XML has become the lingua franca on the Internet, with applications ranging from life sciences [24] to news [22] and advertisement dissemination, among many others. Moreover, XML is changing the way applications exchange messages, the way systems are integrated using service-oriented architectures (SOA), and the way information is stored and processed on the Internet using Web services (WS). Thus, it is not surprising that XML has been widely advocated as data representation for selective information dissemination applications and for content-based message routing [23, 6, 5, 12, 3, 13, 19]. In these scenarios, it is expected that the filtering engines can be capable of processing several millions of filter expressions over high XML document input rates.

For selective information dissemination applications (SID), a filter expression designates an entity's (user or system) interest to receive information of the specified nature. The XML document constitutes the information content to be selectively disseminated. XPath expressions have been proposed for modeling such filters [23, 6, 5, 12, 11, 3, 13, 4, 21, 15, 19]. However, the potential for exploiting overlap in XPath expressions enabled by current filtering techniques is mostly limited to prefix overlap, which is but one source of overlap (i.e., XPath expressions may have infix or suffix overlap as well.) An XPath expression can specify filter constraints on the document's structure, its attributes, and its content. Filter selectivity on a very fine-grained basis is thus possible. Furthermore, the existing algorithms are mostly limited to NFA- or DFA-based processing models, although expressive, this choice substantially hinders the overall performance, as we demonstrate experimentally.

It is this filtering problem that we set out to solve by looking at it from the angle of a publish/subscribe matching problem defined over Boolean predicates (i.e., XPath Expression) and sets of attribute-value pairs (i.e., XML document.) Much research has gone into developing efficient content-based publish/subscribe matching algorithms [27, 10, 1, 7, 2, 26]. All these approaches assume neither XML nor XPath. In this paper, we present the design and implementation of a novel matching algorithm that leverages these existing publish/subscribe matching algorithms and exploits their strengths for solving the XML/XPath filtering problem. We introduce GPX-Matcher (**G**eneric **P**redicate-based **X**Path **M**atcher), essentially a novel generic encoding of XPath expressions into conjunctions of Boolean predicates and of XML documents into sets of attribute-value pairs together with a selectivity-based filtering data structure. Furthermore, our XPath/XML encoding is directly applicable to the existing rich predicate-based matching algorithms (e.g. [27, 7, 2, 26, 25]).

Publish/subscribe matching algorithms are designed to ex-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT 2011, March 22–24, 2011, Uppsala, Sweden.
Copyright 2011 ACM 978-1-4503-0528-0/11/0003 ...$10.00

ploit common predicates and subscriptions by storing and evaluating them only once, thus, considerably reducing storage requirements and processing time. The base assumption is that for SID applications there is always a considerable amount of overlap in the subscription population. Moreover, highly-optimized processing schemes, such as the prefetching technique, clustering, and the introduction of access predicates, developed in [7] and table-based processing of predicates over reasonably-sized, finite domains developed in [2], have proven to lead to significant gains in this context. Thus, an XML filtering technique that adapts these concepts, as well as other directions such as the ones proposed in [27, 26], directly benefits from these concepts.

Another major benefit of our encoding is that it could be directly applied by emerging hardware-based filtering engines, such as fpga-ToPSS [25]. GPX-Matcher's regular, more tabular organization, similar to [7], as well as Boolean expression and attribute-value pairs-based nature, lends itself well to processing in hardware, thus, further boosting performance.

Finally, another advantage is that a solution cast as a publish/subscribe matching problem can immediately be applied to the problem of matching in distributed, content-based publish/subscribe systems (e.g., applied in [8, 14, 20] for the processing of XML/XPath.) Thus, GPX-Matcher can be used to support content-based XML message routing in a distributed information dissemination system, where matching is performed as a routing mechanisms requiring no functional or conceptual changes.

As part of this work, we present X-ToPSS, the XML-based Toronto Publish/Subscribe System [13, 19], which is a fully functional end-to-end matching engine prototype based on GPX-Matcher that has been designed with the above application scale in mind (i.e., processing of several million filter expressions over high document rates.) In this paper, we report on experiments with matching times in the millisecond range for up to four million XPath filter expressions. Since our objective is an end-to-end system, XML document parsing time cannot be ignored in the overall evaluation, as it is common practice in related approaches. For example [6] reports that *"document parsing is another fixed overhead"* and *"further improvements in path navigation time will have at best, a minor impact on overall performance"*. However, we find that parsing is not a dominating factor in our implementation and for the experiments we ran, and is thus negligible in the overall evaluation.

In brief, the main contribution of this paper is a novel encoding and data structure for solving the XML/XPath matching problem applicable for building high-performance XML selective information dissemination applications. Our approach is capable of structural matching and value-based attribute filtering. Furthermore, we show how to advantageously choose access predicates to improve the performance of our data structure, a critical point not addressed in prior work. We perform a thorough experimental analysis that evaluates the performance of our algorithm under varying conditions and present a detailed trade-off analysis of our algorithm against proven filtering techniques.

In the rest of this paper we adopt the terminology common in content-based publish/subscribe and, when appropriate, we will refer to XPath expressions as subscriptions over the constituent Boolean predicates and XML documents as publications of attribute-value pairs.

This paper is organized as follows. Section 2 reviews related work. In Section 3 we present our encoding to express XML documents as attribute-value pairs and express XPath expressions as conjunctions of Boolean predicates. Section 4 presents our matching algorithm operating with the encoded representation in detail. Section 5 presents a detailed performance evaluation of our approach and establishes trade-offs between our approach and the alternative algorithms.

2. RELATED WORK

The related work can be broadly classified into publish/subscribe matching algorithms and XML/XPath matching algorithms We discuss these areas in turn.

Much work has been devoted to the development of matching algorithms in the context of publish/subscribe systems. Our work builds on these algorithms [27, 10, 1, 7, 2, 18, 26]. Our design enables the techniques developed in this body of work to be directly applied to XPath/XML matching. However, none of these algorithms addresses techniques to process XML message against XPath expressions or even hints at tree-structured data processing. All the above approaches assume sets of attribute-value pair as publications and conjunctive Boolean predicate formulas as subscriptions.

Many related approaches have looked at the XML/XPath matching problem [23, 6, 5, 12, 11, 3, 4, 13, 19, 16, 15, 21]. However, none of these approaches is pursuing an approach of reducing the matching problem to the problem of matching with attribute-value pairs in order to leverage existing matching techniques, with the exception of [23, 13]. On the contrary, the above listed approaches introduce new techniques that can be broadly classified into automaton-based algorithms and index-based ones.

The automaton-based approaches [6, 11, 12, 4], most prominently YFilter [6], build automata based on the XPath expressions in the system, while [4] also employs an NFA pruning strategy based on prefix and suffix overlap of XPath expressions. Other automaton-based approaches mostly focus on twig patterns. pFiST introduces value-based filtering [16], iFist proposes a holistic matching of twig patterns using a bottom-up approach [15], and BoXFilter also uses a bottom-up approach, but uses the Prüfer sequence for sequencing twig patterns [21].

The index-based approaches [23, 3, 5, 13] take advantage of precomputed schemes on either the XML documents or the XPath expressions. The index-based approaches are further extended by Huo and Jacobsen [13] toward the idea of predicate-based encodings. However, this approach heavily relies on special-purpose data structures employing a complicated ordering semantics to maintain predicate interrelationships during the matching process [13]. Also, the encoding chosen differs from the one developed in this paper. In our evaluations, we show that GPX-Matcher outperforms both BPA (called basic-pc-ap in [13]) and the automaton-based Yfilter [6].

The WebFilter project [23] experimented with the use of publish/subscribe algorithms for XML/XPath filtering and constitutes an early demonstration of the ideas of XML data dissemination, but does not describe the detailed workings of the approach.

An approach based on string matching to assist in computing XML message routing decisions in the context of distributed content dissemination systems is developed in [19, 17]. However, the techniques developed do not resemble any of the above discussed approaches. The core of the approach is based on string matching path expressions to determine matching candidates.

Next, we review YFilter in more detail, as we use it in our trade-off analysis in the evaluation section of this paper. YFilter is a well-known algorithm for matching XML documents against XPath expressions. It builds a non-deterministic finite automaton (NFA) from all the XPath expressions in the system. The final states of the NFA are associated with list of expressions. The parsing of the XML message, one tag at a time, triggers the transitions in the NFA. Whenever a final state is reached, it means that the corresponding expressions are matched by the incoming XML message. The only difference between a traditional NFA and the YFilter data structure is that the execution of the NFA does not stop when the first final state is reached, but it continues until all possible accepting states are visited. Thus, it determines all matching queries.

3. XML AND XPATH ENCODINGS

Our GPX-Matcher algorithm translates all XPath expressions to a predicate calculus that specify path-constraint predicates. These predicates are evaluated over sets of attribute-value pairs comprising XML document paths.

The objective of this approach is to be able to reduce the problem of XML/XPath matching to the problem of matching attribute-value pairs in the core matching logic of the filtering engine. Section 4 will illustrate this point further.

3.1 XPath expression encoding

Each XPath expression is represented as a conjunction of predicates, which are evaluated over document paths. In what follows, we first present the types of predicates that are required to support such an encoding, and then we show how various XPath expression language features are represented in our predicate calculus.

3.1.1 The predicate calculus

Our predicate calculus supports predicates which express constraints over the absolute and relative position of tags in the document path. In addition, our XPath encoding grows linearly with respect to the size of an XPath expression, i.e., the number of tags, wildcards and descendant operators, because in our encoding each tag in XPath expression is represented by at most a single predicate in the encoding.

The predicate (p_t *op* v), where v is a natural number, represents a constraint on the position of the tag t in the XML document path. This is a predicate over one free variable p_t. The operator *op* can be either $=$ or $\geq$. This predicate is satisfied for a given document path if and only if the path contains the tag t at position v', which satisfies the relation v' *op* v. This predicate is used to represent a constraint on the position in the XML path of the first location step in the XPath expression that is not a wildcard.

The predicate ($p_t^{\dashv} \geq v$) represents a constraint on the position of the tag t relative to the end of the XML document path. This predicate is satisfied for a given document path of length l if and only if the path contains the tag t at position v' such that $l - v' \geq v$. This predicate is used to represent constraints for XPath expressions that end in wildcards.

The predicate ($d(p_{t_1}, p_{t_2})$ *op* v) represents a predicate over two free variables that imposes a constraint on the relative position of the tag t_2 to the tag t_1. The operator *op* can be either $=$ or $\geq$. This predicate is satisfied for a given document path if and only if the path contains both tags t_1 and t_2 at positions v_1 and v_2, respectively, and these positions satisfy the relation $(v_2 - v_1)$ *op* v. In other words, the tags are situated at a distance d from each other such that d *op* v holds. Note that the order between the tags t_1 and t_2 in the path is important. The order determines the sign for d. This type of predicate is used to encode constraints on consecutive location steps that are not wildcards.

The last type of predicate is a special one that does not refer to tag names or their positions, but to the length of the XML document path. The predicate ($length \geq v$) represents a constraint on the length of the XML document path. This predicate is satisfied for a given document path if and only if the length of the path is greater than v. This special type of predicate is required to represent XPath expressions that contain only wildcards.

3.1.2 XPath expressions

Next, we show how our predicate calculus is used to encode XPath expressions. In order to distinguish between different tags with the same name, each tag name is associated with its occurrence number. For example, the XPath expression $/a/b/c/a$ is represented as $/a^1/b^1/c^1/a^2$, where the superscripts represent the occurrence numbers.

Simple XPath expressions: We call an XPath expression *simple* if it does not contain wildcards (*) or descendant operators (//). The relative XPath expression $s = t_1/.../t_n$ (all t_k represent tag names other than "*") is mapped to the following conjunction of predicates:

$$(p_{t_1} \geq 1) \wedge (d(p_{t_1}, p_{t_2}) = 1) \wedge \cdots \wedge (d(p_{t_{n-1}}, p_{t_n}) = 1)$$

The absolute XPath expression $s = /t_1/.../t_n$ is mapped to the following conjunction of predicates:

$$(p_{t_1} = 1) \wedge (d(p_{t_1}, p_{t_2}) = 1) \wedge \cdots \wedge (d(p_{t_{n-1}}, p_{t_n}) = 1)$$

The conjunction contains as many predicates as steps in the XPath expression. The first predicate captures the position of the first tag in the XML document path, while all other predicates represent constraints for the relative position of the current tag to the previous one. For simplicity, in the conjunction for relative expressions, the first predicate can be discarded. This is possible because the second predicate requires that t_1 be in the path and once the path contains a tag name, then its position is greater or equal to one.

Wildcards in XPath expressions: Consider the following relative XPath expression (all t_k with $1 \le k \le n$ and $k \le i$ or $k \ge i+m+1$ represent tag names other than "*"):

$$s = t_1/.../t_i/*/.../*/t_{i+m+1}/.../t_n$$

with m consecutive wildcards starting at location step $(i+1)$. This expression is mapped to predicates by bypassing the wildcard steps, as follows:

$$((d(p_{t_1}, p_{t_2}) = 1) \quad \wedge \cdots \wedge \quad (d(p_{t_{i-1}}, p_{t_i}) = 1) \;\wedge$$
$$(d(p_{t_i}, p_{t_{i+m+1}}) = m+1) \quad \wedge \cdots \wedge \quad (d(p_{t_{n-1}}, p_{t_n}) = 1))$$

The representation of an absolute XPath expression is similar to the relative one (it contains one more predicate to restrict the position of the first tag name.) The number of predicates required to represent an XPath expression with m consecutive wildcards is m predicates less than the number of predicates required for representing the expression without wildcards.

To generalize, consider the following XPath expression:

$$s = xpe_1/*/.../*/xpe_2/*/.../*/xpe_3.../*/.../*/xpe_n$$

where xpe_k represent simple XPath expressions and the ith sequence of wildcards contains m_i steps. This expression is translated into a conjunction of predicates that contains all predicates that represent the simple expressions xpe_k $(1 \le k \le n)$ and all predicates p_i $(1 \le i < n)$ that link the position of the last tag in xpe_i and the position of the first tag in xpe_{i+1}: $d(p_{last_tag_in_xpe_i}, p_{first_tag_in_xpe_{i+1}}) = m_i+1$ (we call these predicates *link predicates*.)

There are three special cases that need to be addressed in the representation of s:

(1) The expression starts with m wildcards: the overall conjunction contains one more predicate that represents a constraint for the position of the first tag in xpe_1: $(p_{first_tag_in_xpe_1}\ op\ m+1)$. The operator is determined by the type of the XPath expression: a relative expression requires the *greater than* operator, while an absolute one requires the *equal* operator.

(2) The expression ends with m wildcards: the overall conjunction contains one more predicate that represents a constraint for the relative position of the last tag in xpe_n to the end of the XML document path: $(p^{\dashv}_{last_tag_in_xpe_n} \ge m)$.

(3) The expression contains only m wildcards: the representation contains only one predicate: $(length \ge m)$.

Descendant operators in XPath expressions: Consider the following relative XPath expression:

$$s = t_1/.../t_i//t_{i+1}/.../t_n$$

with a descendant operator after location step i. This expression is mapped to predicates as follows:

$$(d(p_{t_1}, p_{t_2}) = 1) \quad \wedge \cdots \wedge \quad (d(p_{t_{i-1}}, p_{t_i}) = 1) \;\wedge$$
$$(d(p_{t_i}, p_{t_{i+1}}) \ge 1) \quad \wedge \cdots \wedge \quad (d(p_{t_{n-1}}, p_{t_n}) = 1)$$

The representation of an absolute XPath expression is similar to the relative one (it contains one more predicate to restrict the position of the first tag name.) The only difference to the representation of a simple expression is the operator for the second order predicate that relates the tags t_i and t_{i+1}.

To generalize, given the following XPath expression:

$$s = xpe_1//xpe_2//xpe_3...//xpe_n$$

where xpe_k represent simple XPath expressions. This expression is translated into a conjunction of predicates that contain all predicates that represent the simple expressions xpe_k $(1 \le k \le n)$ and all predicates p_i $(1 \le i < n)$ that link the position of the last tag in xpe_i and the position of the first tag in xpe_{i+1}: $d(p_{last_tag_in_xpe_i}, p_{first_tag_in_xpe_{i+1}}) \ge 1$ (we call these predicates *link predicates*.)

The above encoding remains unchanged when the expressions xpe_k contain wildcards that are neither at the beginning nor at the end of the expressions. For expressions xpe_k that start or end with wildcards, only the link predicates change. Suppose there are two expressions xpe_l and xpe_{l+1} such that xpe_l ends with m wildcards and xpe_{l+1} starts with n wildcards, then the lth link predicate will have to change to $d(p_{last_tag_in_xpe_l}, p_{first_tag_in_xpe_{l+1}}) \ge m+n+1$.

All encodings presented above are unique in the sense that two different XPath expressions have different encodings and the same encoding corresponds to identical subscriptions.

3.1.3 Examples

The following exemplifies our XPath expression encoding.

X_1 : /b/c
S_1 : $(p_{b^1} = 1) \wedge (d(p_{b^1}, p_{c^1}) = 1)$
X_2 : a/a/*/b
S_2 : $(d(p_{a^1}, p_{a^2}) = 1) \wedge (d(p_{a^2}, p_{b^1}) = 2)$
X_3 : /a/a/*/*
S_3 : $(p_{a^1} = 1) \wedge (d(p_{a^1}, p_{a^2}) = 1) \wedge (p^{\dashv}_{a^2} \ge 2)$
X_4 : /*/*/*/a/b
S_4 : $(p_{a^1} = 4) \wedge (d(p_{a^1}, p_{b^1}) = 1)$
X_5 : a//b/c
S_5 : $(d(p_{a^1}, p_{b^1}) \ge 1) \wedge (d(p_{b^1}, p_{c^1}) = 1)$
X_6 : a/b/*//a
S_6 : $(d(p_{a^1}, p_{b^1}) = 1) \wedge (d(p_{b^1}, p_{a^2}) \ge 2)$
X_7 : a/b/*/*//*/a
S_7 : $(d(p_{a^1}, p_{b^1}) = 1) \wedge (d(p_{b^1}, p_{a^2}) \ge 4)$
X_8 : x/a/b/*/*//*/a/*
S_8 : $(d(p_{x^1}, p_{a^1}) = 1) \wedge (d(p_{a^1}, p_{b^1}) = 1)$
$\wedge (d(p_{b^1}, p_{a^2}) \ge 4) \wedge (p^{\dashv}_{a^2} \ge 1)$

Some of the predicates above are duplicated due to overlap in the XPath expressions. However, in our data structure, we represent each predicate only once (i.e., no duplicated predicates are stored or processed.) During the matching phase of our algorithm, predicates that represent multiple overlapping XPath expressions are evaluated only once for all the expressions that contain them. For example, consider XPath expressions S_7 and S_8. The total number of predicates for expressing these two expressions is six (i.e., two for S_7 and four for S_8), but the number of distinct predicates is four, since all predicates of S_7 are in S_8 as well. This is

one of the strengths of our algorithm, as it resolves most overlaps in XPath expressions and represents and processes these parts only once.

3.1.4 Attribute-based Filters on XPath Expressions

In our discussion so far we referred only to XPath expressions that do not contain filters on the attributes of the XML document or nested paths. Our predicate calculus can be easily extended to support filters on the attribute content. Consider a location step that contains the tag name t and the predicate [@ *attr op value*]. In order to represent this filter, we introduce a new type of predicate in our system, $d_t[attr\ op\ value]$, that expresses a constraint on the value of the attribute *attr* attached to the tag name t in the XML document. The publications resulting from the XML paths are augmented with information about the values of all attributes of a tag name t. This is also evaluated in Section 5

Nested path filters can be treated in a manner orthogonal to the matching algorithm presented in this section. A scheme similar to that in [6] could be used in our system in order to support nested paths.

3.2 XML document encoding

Let d be an XML document. Document d consists of a number of paths, $d = (e_1, ..., e_l)$, from the root element of the document to each leaf. A path e_i consists of the tag elements, attributes and the corresponding values, and content elements along the path from the root element to a leaf:

$$e_i = \{t_1^i, [(a_{t_1,1}^i, v_{t_1,1}^i), ..., (a_{t_1,j_1}^i, v_{t_1,j_1}^i)], c_1^i, ...,$$

$$t_n^i, [(a_{t_n,1}^i, v_{t_n,1}^i), ..., (a_{t_n,j_n}^i, v_{t_n,j_n}^i)], c_n^i\}$$

where $a_{t_l,k}^i$ is the kth attribute associated with the lth tag in the ith path of d, $v_{t_l,k}^i$ represents the corresponding value for the attribute and c_l^i represents the content of the lth tag. For brevity sake, we simply write $e = (t_1, ..., t_n)$; the subscript of each tag represents the position of the tag in the path. In this formalization, we do not explicitly show content and attribute elements, and since any path can contain duplicate elements, each tag name has an occurrence number associated (not shown in our notation above.)

First, let us consider the case when a path $e = (t_1, ..., t_n)$ contains only distinct tag names (i.e., all occurrence numbers are equal to one.) This path is translated to the following set of tuples:

$$\begin{array}{cc} (length, n) & , \\ (t_1, 1), (t_2, 2), \ldots (t_n, n) & , \\ (t_1, t_2, 1), (t_1, t_3, 2), ..., (t_1, t_n, n-1) & , \\ (t_2, t_3, 1), (t_2, t_4, 2), ..., (t_2, t_n, n-2) & , \\ ... & \\ (t_{n-1}, t_n, 1) & \end{array}$$

This set contains T(n)=$O(n^2)$ tuples. This encoding is dictated by our predicate calculus. The length of the path is needed to check predicates of type $(p_t^{\dashv} \geq v)$ and $(length \geq v.)$ All tag positions are required to verify predicates of type (p_t op v) and $(p_t^{\dashv} \geq v)$. Finally, all combinations of two tags are required in order to verify predicates of type $(d(p_{t_1}, p_{t_2})\ op\ v)$. We call this set of tuples a *publication*. Each publication is a set of attribute-value pairs, where the attribute name can be the length $(length \geq v)$, any tag name (p_t op v), and any combination of two tags $(d(p_{t_1}, p_{t_2})\ op\ v)$, while the value is a natural number less than or equal to the length of the XML path.

Before discussing the case when the path contains duplicates, we need to present some notational conventions. Given a path $e = (t_1, ..., t_n)$, we refer to a *subpath* of e which contains a subset of the tags in e ordered by their positions in the path and write $s_e = ((k_1, t_{k_1}), (k_2, t_{k_2}), ..., (k_p, t_{k_p}))$ (k_i represents the position of the tag t_{k_i} in the path e.) The publication $p(s_e)$ resulting from a subpath $s_e = ((k_1, t_{k_1}), (k_2, t_{k_2}), \cdots, (k_p, t_{k_p}))$ contains the following tuples:

$$\begin{array}{cc} (length, k_p - k_1 + 1) & , \\ (t_{k_1}, k_1), (t_{k_2}, k_2), \ldots (t_{k_p}, t_{k_p}) & , \\ (t_{k_1}, t_{k_2}, k_2 - k_1), (t_{k_1}, t_{k_3}, k_3 - k_1), ..., (t_{k_1}, t_{k_p}, k_p - k_1) & , \\ (t_{k_2}, t_{k_3}, k_3 - k_1), (t_{k_2}, t_{k_4}, k_4 - k_2), ..., (t_{k_2}, t_{k_p}, k_p - k_1) & , \\ ... & \\ (t_{k_{p-1}}, t_{k_p}, k_p - k_{p-1}) & \end{array}$$

We write $p(e - \{t_{l_1}, t_{l_2}, ..., t_{l_p}\})$ to denote the publication resulting from the subpath of e after eliminating the tags $t_{l_1}, t_{l_2}, ..., t_{l_p}$. Note that after eliminating some tags that have duplicates, the occurrence numbers are adjusted as needed. For example, consider the path $e = (a, b, c, a, b)$. Including the occurrence numbers, the path is expressed as $e = (a^1, b^1, c^1, a^2, b^2)$ (the superscripts represent the occurrence numbers.) The subpath of e after eliminating the first a tag, written as $s_e = e - \{a^1\}$ is represented as $s_e = ((2, b^1), (3, c^1), (4, a^1), (5, b^2))$. Note that when eliminating a^1, the occurrence number of the second occurrence of a is adjusted to 1.

Consider the case when the path contains only one tag t that is duplicated d times (i.e., the same tag name t in d different positions in the path.) The corresponding occurrence numbers for this tag are 1, 2, ..., and d, respectively. Since a tag t with the occurrence number k from an XPath expression can be mapped to any tag t from the XML path with the occurrence number $o \geq k$, we have to map the XML path to a set of publications as follows:

$$P(e) = \bigcup_{S \in P(T)} \{p(e - S)\}$$

where $T = \{t^1, t^2, ..., t^{d-1}\}$, $P(T)$ is the power set of T, and $e - S$ is the subpath obtained from e after eliminating the tags in S. The corresponding occurrence numbers are adjusted in the subpath as explained above. We do not need to look at the subpaths in which t^d is eliminated because the elimination of the dth occurrence of the tag will not change the occurrence numbers. Thus, it does not add any new information to the publication. Generally speaking, if the path e contains tags $t_{k_1}, t_{k_2}, \ldots, t_{k_r}$ with the corresponding number of duplicates $d_{k_1}, d_{k_2}, \ldots, d_{k_r}$, the path e is mapped to the following set of publications:

$$P(e) = \bigcup_{S \in P(T)} \{p(e - S)\}$$

where $T = \bigcup_{l=1}^{r} \{t_{k_l}^1, t_{k_l}^2, ..., t_{k_l}^{d_{k_l}-1}\}$, $P(T)$ is the power set of T and $e - S$ is the subpath obtained from e after eliminating the tags in S (the occurrence numbers are adjusted accordingly.)

e:	$((1, a^1), (2, b^1), (3, b^2), (4, c^1))$
	$((length, 4), (a^1, 1), (b^1, 2), (b^2, 3), (c^1, 4), (a^1, b^1, 1), (a^1, b^2, 2), (a^1, c^1, 3), (b^1, b^2, 1), (b^1, c^1, 2), (b^2, c^1, 1))$
$e - \{b^1\}$	$((1, a^1), (3, b^1), (4, c^1))$
	$((length, 4), (a^1, 1), (b^1, 3), (c^1, 4), (a^1, b^1, 2), (a^1, c^1, 3), (b^1, c^1, 1))$

Table 1: Subpath, representation and publication.

Usually, XML paths contain a small number of duplicate tags and the occurrence numbers are also small. For the case when the path contains only one tag with two duplicates, there are only two publications built. The total number of tuples built still remains on the order of $O(n^2)$.

To summarize, each path $e = (t_1, ..., t_n)$ is translated to one or several publications; each publication contains a set of attribute-value pairs. Thus, the XML document d is represented as a set of publications: $d = (pub_1, ..., pub_p)$ such that each path generates at least one publication and any path that contains duplicate tags generates more than one publications.

3.3 Matching XML against XPath expressions

Next, we explain how the matching problem of XML documents against XPath expressions is reduced to the problem of matching publications against conjunctions of predicates. First, let us present how each type of predicate from our calculus is matched by a publication:

- A predicate of type $(length \geq v)$ is satisfied by a publication pub if the value v' in the tuple $(length, v')$ from the publication pub satisfies the relation $v' \geq v$.
- A predicate of type $(p_t\ op\ v)$ is satisfied by a publication pub if the publication contains a tuple (t, v') such that v' satisfies the relation $v'\ op\ v$.
- A predicate of type $(p_t^{\dashv} \geq v)$ is satisfied by a publication pub if the publication contains a tuple (t, v') such that $l - v' + 1 \geq v$, where l is the value from the tuple $(length, l)$ in pub.
- A predicate of type $(d(p_{t_1}, p_{t_2})\ op\ v)$ is satisfied by a publication pub if the publication contains a tuple (t_1, t_2, v') such that v' satisfies the relation $v'\ op\ v$.

  In all instances above, the matching between tag names also respects the occurrence numbers.

A publication pub matches a conjunction of predicates $s = (pred_1 \wedge pred_2 \wedge ... \wedge pred_m)$ if all predicates in s are satisfied by the publication.

Formally, an XPath expression is matched by an XML document if and only if the XPath expression selects a non-empty set of nodes from the XML document. Using the encoding described above, the matching problem is formulated as follows: *Given an XML document represented as $d = (pub_1, ..., pub_p)$ and an XPath expression $s = (pred_1 \wedge pred_2 \wedge ... \wedge pred_m)$, the XPath expression is matched by the XML document if and only if there is at least one publication pub_i $(1 \leq i \leq p)$ such that all predicates in s are satisfied by the publication pub_i.*

Predicate	$p(e)$	$p(e - \{b^1\})$
$p_{b^1} = 1$	X	X
$d(p_{b^1}, p_{c^1}) = 1$	X	$\surd$
$d(p_{a^1}, p_{b^1}) \geq 1$	$\surd$	$\surd$

Table 2: Matching publications against predicates

Example – Consider the XPath expressions S_1 and S_5 from the earlier list with their corresponding conjunction of predicates and an XML document that contains the following path $e = (a^1, b^1,\ b^2, c^1)$. Table 1 presents all the subpaths that are considered and the corresponding publications. Table 2 shows which predicates within the XPath expressions are matched by each publication. The expression S_5 is matched by the publication $p(e - \{b^1\})$ since all its predicates are matched. The expression S_1 is not matched because none of the publications satisfies S_1's first predicate.

4. THE MATCHING ALGORITHM

Each incoming XML file is processed, and its paths are translated into sets of publications as previously explained. Each publication is submitted to the matching algorithm separately. The matching algorithm proceeds in two stages: first, each publication is matched against the predicates in the system; second, given this information, the matched XPath expressions are identified. After all publications resulting from the XML document are processed, the matched XPath expressions are collected. Next, we present in detail the data structures and algorithms used for the two matching stages.

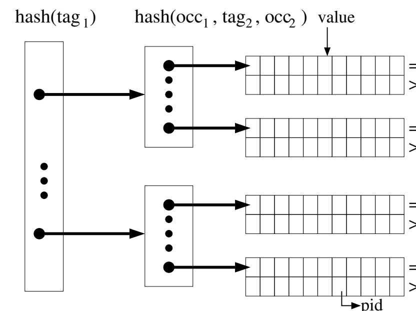


Figure 1: Predicate data structure

4.1 Predicate matching

Each XPath expression is translated into a conjunction of predicates. These predicates are inserted in the corresponding data structures. Recall that our predicate language supports four different types of predicates that refer to tag names and the length of the XML path. The representation of the predicates have to take into account that the tag names may be duplicated in an XPath expression. Thus, each tag name has an associated occurrence number, as already explained. Next, we present the data structures for representing these predicates. Each predicate in our system is represented by a unique identifier.

All values that are used in the predicates are defined over a finite domain $D : [1 : l]$ where l represents the maximum length of the XPath expressions supported by the system. Thus, the main data structure used for storing and evaluating predicates is a predicate-table that extends from 1 to l –

the upper bound of the domain – and can be indexed by the value of the predicate. Each row of the table is uniquely associated to one of the predicate operator (i.e., $=$ or $\geq$). Each element of the table records the corresponding predicate id (pid, for short) or nothing if there is no such predicate in the system. Based on the corresponding operator, the matching performs differently for an attribute-value pair that contains a value v. If the operator is *equality*, then only the predicate identified by the pid at the vth position in the table is set to true. If the operator is *greater than*, then all predicates identified by the pids at positions i, where $i \leq v$, are set to true.

The predicate $(d(p_{t_1}, p_{t_2})\ op\ v)$ represents a second order predicate that imposes a constraint on the relative position of the tag t_2 to the tag t_1. As the tags of the predicate can be any tag names within an XPath expression, each tag name may have an occurrence number greater than one. The data structure used for storing this type of predicates is depicted in Figure 1. The first hashtable is indexed on the first tag name. Each entry of this hashtable points to another hashtable that is indexed on a key composed of the occurrence number of the first tag name, the second tag name, and the occurrence number of the second tag name. Each entry of the second hashtable points to two predicate-tables, one for each operator supported by this type of predicate.

The predicate $(p_t^{\dashv} \geq v)$ represents a constraint on the position of the tag t relative to the end of the XML path. As tag t is the last tag name in the XPath expression, its occurrence number may be greater than one (if the path contains another location step that refers to the same tag name.) The data structure used for storing this type of predicates is also a hashtable on the tag name of the predicate. However, each entry of the hashtable points to a matrix that can be indexed on the occurrence number. Thus, each ith row of the matrix is a predicate-table for the ith occurrence of the corresponding tag name.

The predicate $(p_t\ op\ v)$ refers to the position of the tag in the XML path. This predicate always refers to the first tag name in the XPath expression; therefore, the occurrence number for this tag is always one and the data structure used for these types of predicates does not contain bookkeeping information for occurrence numbers. The data structure used for storing this type of predicates is a hashtable; the key of the hashtable is built using the tag name of the predicate. Each entry of the hashtable points to two predicate-tables, one for each operator supported by this type of predicate.

The predicates that refer to the length of the XML path are stored separately in a predicate-table. Note that the XPath expression that contains a predicate on the length of the XML path is composed only of wildcards. Thus, this is the only predicate that this subscription contains. The matching algorithm treats these subscriptions separately comparing the maximum length of all paths in the incoming XML file against the predicate-table. All matched predicates from this table dictate that the corresponding subscriptions are matched.

Two different XPath expressions may have overlapping predicates (same predicate is contained in both subscriptions.) However, we store the predicate only once (each predicate is unique in the system.) For each predicate we also store a list of all subscriptions that contain that predicate. This information is used in the second stage of matching: subscription matching.

As the paths resulting from the XML documents have overlaps, the publications we build for the paths also have common tuples. This knowledge about which tuple matches which predicates and the evaluation of common tuples in overlapping paths of the XML document only once can be further exploited. This utilization resembles the concept of batch processing proposed in [9], which is complementary to our approach and brings about a new avenue of research to further improve the matching time. Overall, the predicate matching data structures presented above resemble the techniques developed in [2], but have been generalized here to account for the richer predicate calculus required for the encoding of XPath expressions and XML documents.

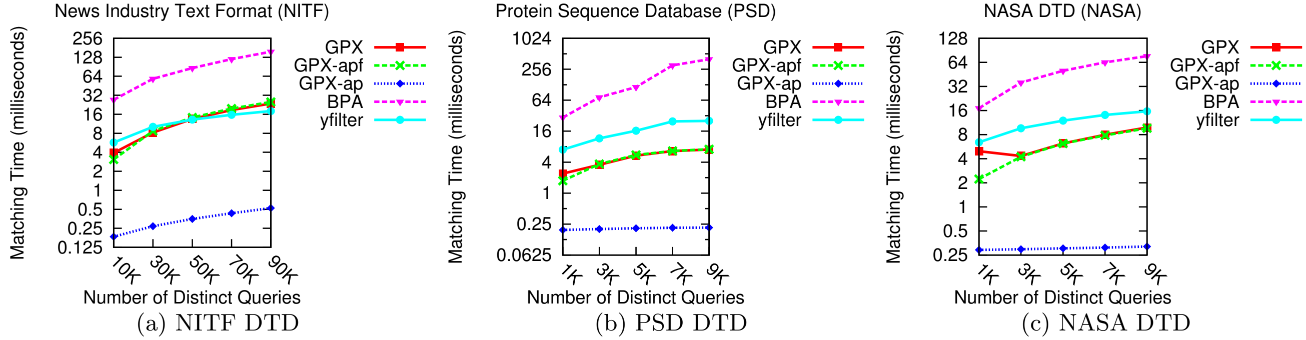
4.2 Subscription matching

Subscription matching represents the second stage of the matching algorithm, in which given all matched predicates, the set of matched subscriptions is built. When an XPath expression (subscription) is added to the system, it is decomposed into its predicates. In addition, we consider only distinct subscriptions in the matching stage. We achieve this by building groups of subscriptions having exactly the same predicates, and we promote only one subscription per group. We call this promoted subscription the group *representative*. Only representatives participate in the actual matching stage. In the final step of the subscription matching, we iterate over the set of matched representatives and set as matched all subscriptions within each group that has its representative matched. For simplicity, we use the term *subscription* and *representative* interchangeably.

Counting algorithm: Here, the subscription matching stage is based on two data structures: the subscription-predicate count vector and the hit vector. The subscription-predicate count vector stores for each subscription the number of its predicates. The hit vector records the number of satisfied predicates per subscription; we call each position in this vector a hit-counter. The hit vector is re-computed for each publication evaluation. During the evaluation of a publication, whenever a predicate is set to true, all hit-counters associated with the subscriptions that contain that predicate are incremented. After all predicates are evaluated, the hit vector is compared against the subscription-predicate count vector. All subscriptions for which each of the two entries are equal are considered as matched. Note that this is a valid operation because each predicate is evaluated to true at most once during the evaluation of a publication. All hit-counters are reset before evaluating another publication.

After all publications resulting from the XML file are evaluated, the set of matched subscriptions represents the matched XPath expressions.

Access predicates: Subscription evaluation is a time-consuming process since the last stage of the algorithm requires iterating over all distinct subscriptions in the system to check the corresponding counters. Moreover, this operation



(a) NITF DTD (b) PSD DTD (c) NASA DTD

Figure 2: Varying the number of distinct queries (with no duplicates)

happens for each generated publication. Each XML path produces at least one publication. In order to reduce the time spent on this stage of the algorithm, we adapt an optimization inspired by [7]. The optimization involves clustering subscriptions on an *access predicate*. Each cluster contains all subscriptions having a common access predicate. If there are subscriptions that contain more than one access predicate, they are inserted only into one cluster. During matching, if a cluster's access predicate is not satisfied, then none of the cluster's subscriptions will be matched. Thus, in the last phase of the algorithm, we have to iterate only over subscriptions whose access predicates are matched. Consequently, These access predicates have to be selective enough such that the number of subscriptions to be checked in the last phase of the algorithm is considerably reduced. Otherwise, the overhead of bookkeeping clusters may not be justified. Although [7] introduces the idea of access predicates, the authors do not present any insights on how these access predicates can be chosen in general. For instance, computing statistics on workload (as suggested in [7]) could be used to estimate predicate selectivity. But, given our domain, the predicates are built from the XPath expressions, and an XPath expression refers to XML documents that have a tree-like structure which further complicates selectivity estimation. However, a key observation in our domain is that the selectivity of the XPath expression tends to be higher as the number of location steps contained in the expression increases. Therefore, we claim that the predicate built over the last location step with a tag name to be the most selective one. This claim is supported by our extensive experimental evaluation (cf. Section 5.)

5. PERFORMANCE EVALUATION

Experiment setup: We experimentally evaluate the performance of our main counting-based algorithm GPX-Matcher (in short GPX) and the selectivity-aware version GPX-Matcher-ap (in short GPX-ap), that is, based on the notion of access predicates. To better understand the performance and the trade-offs of these algorithms, we also compare them against YFilter [6] and BPA [13].

In our X-ToPSS framework, all algorithms are implemented in C. For XML parsing, we implemented a parser based on libxml2. All reported experiments were run on an Intel Quad-Core 2.66GHz with 4GB of main memory running Ubuntu 10.4. Since our implementation is not multi-threaded, none of the algorithms take advantage of more than a single core.

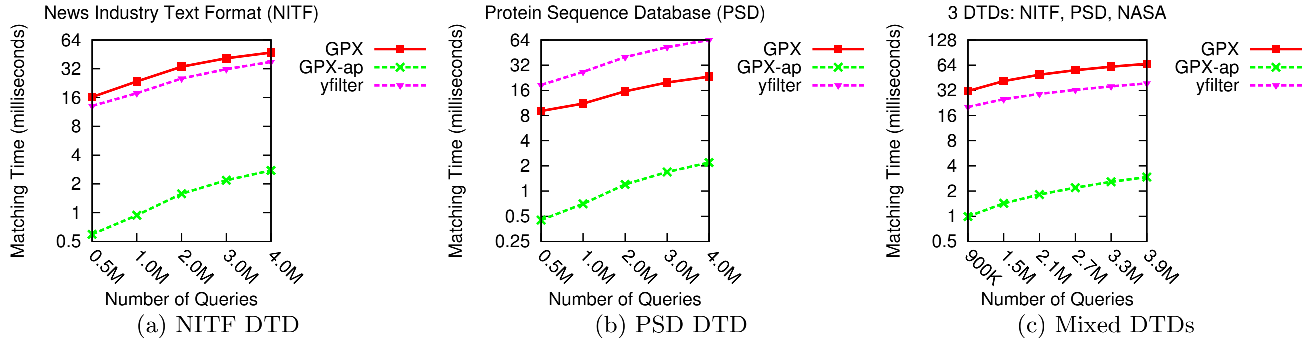
None of the algorithms require or exploit DTD information. However, we used DTDs to generate workloads for our experiments. The DTDs we used are: NITF (News Industry Text Format), PSD (Protein Sequence Database) and NASA. These DTDs are commonplace, and are used as standard benchmarks in many research studies such as [12, 6, 5]. Due to space limitations, we included only selected results obtained via the NASA DTD.

In order to generate the XPath expressions, we used the XPath generator released by Diao *et al.* [6]. We control the XPath workload in the experiments through the following parameters: Q represents the number of queries ranging from $25,000$ to $4,000,000$, D indicates whether the queries are distinct, L represents the maximum length of the XPath queries ranging from 6 to 12, W represents the probability of a wildcard occurring at a location step (varied between 0-0.7), DO represents the probability of a descendant operator at a location step (again varied from 0-0.9), and P represents the number of predicate filters per query ranging from 1 to 10.

For generating the XML documents, we relied on the IBM XML Generator. The XML generator was used with the default parameters with only one exception: we varied the maximum number of levels in the resulting XML tree from 6 to 12. This guarantees that if the resulting tree goes beyond the maximum number of levels, the generator will add none of the optional elements (denoted by * or ? in the DTD) and only one of each of the required elements (denoted by + or no option.) As a result, the number of levels of the resulting XML files may be greater than the actual value of this parameter.

For each DTD, we generated 500 XML documents. All reported results are averaged over this set, unless otherwise stated. For each experiment, a set of XPath expressions was generated using the specified parameters. For each algorithm, we filtered the 500 XML files against the query workload, then we averaged the resulting time. Note that the time to build the predicate data structures or the NFA (for YFilter) is not included in the filtering time.

All our experiments use the total filtering time as the main performance metric. This time includes parsing the XML documents, matching them against the query workload, and collecting the results. For all studied algorithms, the matching result is represented as a bit vector with one bit for each



(a) NITF DTD (b) PSD DTD (c) Mixed DTDs

Figure 3: Varying the number of queries (with duplicates)

query; a query is matched if and only if its corresponding bit is set to 1. As our experiments will show, parsing time is negligible compared to the total filtering time. The average parsing time for NITF and PSD XML documents are on the order of microseconds, while the total matching time are on the order of millisecond.

Varying the number of queries: We begin the performance evaluation by looking at scalability of the algorithms as we vary the number of queries. First, we look at distinct queries, and then at workloads that contain duplicate queries.

Distinct Queries: The query workload is generated using $L = 6$, D=true, $W = 0.2$, and $DO = 0.2$. The number of distinct queries is varied from $10,000$ to $90,000$ for the NITF DTD and from $1,000$ to $9,000$ for the PSD DTD. The PSD DTD limits the number of distinct queries that can be generated with the above parameters, since it is not recursive and the maximum number of levels is 8. The results for the NITF DTD are shown in Figure 2(a). As seen, GPX-ap not only gracefully scales as the number of distinct queries is increased, it also exhibits superior performance; that is, filtering $90,000$ XPath expressions takes just under 1 ms, as GPX-ap fully utilizes the workload selectivity, compared to 18 ms for YFilter. In fact, in the NITF workload, the generated queries are highly selective, in which the percentage of matched queries on average is about 0.06. On the one hand, GPX-ap, fully exploits workload selectivity through its novel access predicate selection (only possible due to the unique encoding it employs) and dominates all other approaches (GPX-apf, a variation of GPX-ap, performance is discussed in more detail later.) On the other hand, as expected, YFilter also performs better than our baseline GPX and BPA as the number of distinct queries increases because YFilter also, to a limited degree, utilizes the workload selectivity by reducing the number of the NFA's triggered states.

Figure 2(b) presents the results for the PSD DTD. Most importantly, the difference between YFilter and GPX becomes more significant in this case. For $10,000$ queries, YFilter takes more than three times as much time as GPX. This happens because the selectivity is lower. Therefore, the execution of the NFA takes more time as more of its states are touched during filtering. The smaller number of queries translates into a smaller number of predicates for GPX. Lower selectivity also slightly worsens the performance of GPX-ap, yet it remains dominant. The NASA workload with results in Figure 2(c) also follows the same pattern as the PSD DTD results.

Finally, the detailed distinction between the GPX-ap and GPX-apf algorithms is postponed to a later section that is dedicated to predicate clustering. For the rest of our experiments, we shift our focus to the top performing algorithms: GPX, GPX-ap, and YFilter.

Queries with duplicates: Large filtering systems (e.g., news filtering and dissemination services) are characterized by large subscription workloads that are likely to contain duplicate and overlapping subscriptions. These duplicates represent common interests of different subscribers. In order to study the effect of duplicate subscriptions on filtering time, in this experiment, we vary the number of queries from half a million to four million and set D to false (the query generator will not eliminate duplicates), while adopting the other parameter values from our previous experiment. The results are summarized as follows: Figure 3(a) based on the NITF workload, Figure 3(b) based on the PSD workload, and Figure 3(c) based on a mix of the NITF, PSD, and NASA workloads. For all DTDs, all algorithms scale gracefully as the number of queries increases. Note, again, that YFilter is outperformed even by our baseline GPX for the PSD workload. Most prominently, our GPX-ap algorithm significantly improves over all algorithms across all populations of queries by an order of magnitude.

Predicate clustering: As we have observed so far (Figure 2(a-c)), the predicate clustering implemented by GPX-ap improves the total filtering time by an order of magnitude in comparison with our baseline approach GPX. This supports our claim that parsing is not a dominant factor in the overall matching time. In fact, if the parsing time was dominant, then no significant gain was attainable by using our GPX-ap. Next, we want to further explore our various predicate clustering approaches, i.e., GPX-apf and GPX-ap.

First, we validate through experiments the intuition that the last predicate of each XPath expression is a good candidate as an access predicate. We implement clustering on both the first and on the last predicate representing the XPath expressions. Figure 2(a-c) shows the results for these experiments using different DTDs, where clustering on the first predicate is labeled GPX-apf. As expected, the clustering on the first predicate of each query provides no additional gain as it fails to exploit the workload's underlying selectiv-

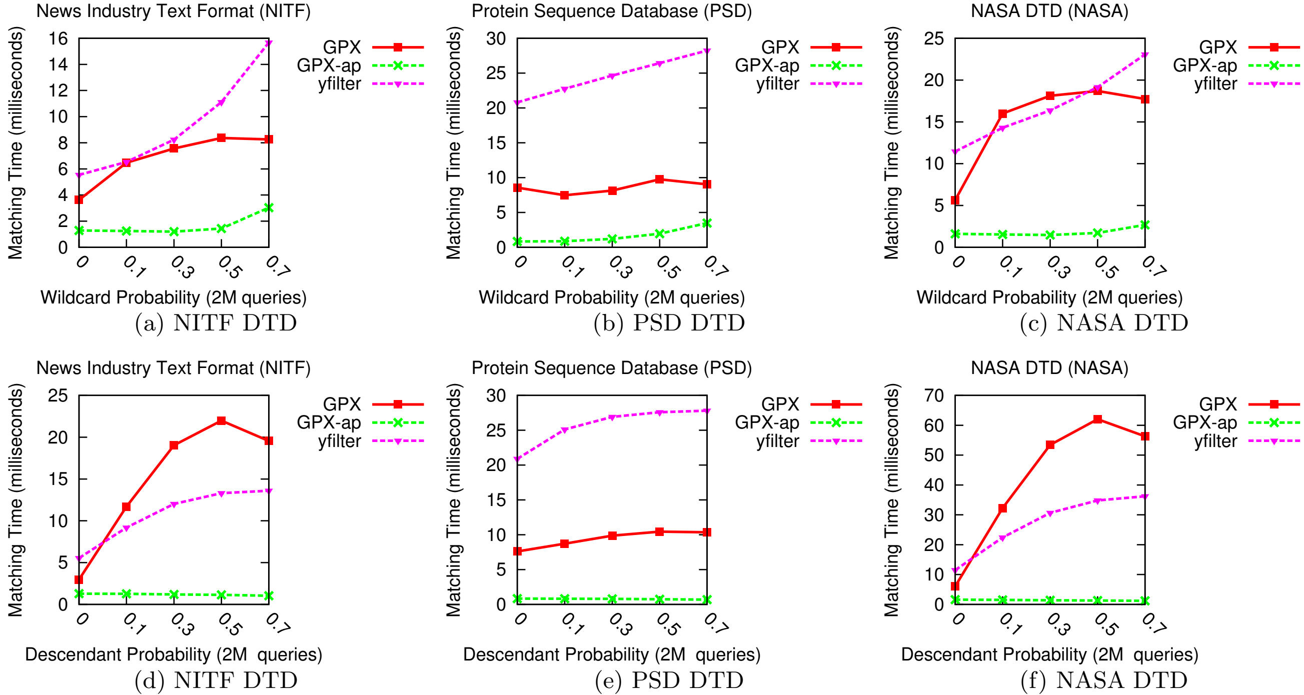


Figure 4: (a)-(c) Varying wildcard (*) probability (d)-(f); Varying descendant (//) probability

ity. Therefore, the minor overhead incurred for managing the clusters could potentially deteriorate the performance as well because it provides no pruning and nearly all subscriptions are checked. In contrast, GPX-ap's effectiveness is clearly evident which results in substantial performance gain and improves the performance by considerably reducing the number of subscriptions to be checked for matching.

Effects of wildcards and descendant operators: We investigate the impact of the probability of wildcards (W) and descendant operators (DO). We built two different query workloads. First, we set $DO = 0.2$ and varied W from 0-0.7; second, we set $W = 0.2$ and varied DO from 0-0.7. All workloads have two million queries. The results of varying the wildcard probability for NITF, PSD, and NASA workloads are demonstrated in Figure 4(a), 4(b), and 4(c), respectively.

Increasing the wildcard probability has an interesting effect, that is, it decreases the query selectivity. For instance for the NITF dataset, when $W = 0$, a common XPath expression of length 6 is:

/nitf/body/body.head/distributor/org/alt-code.

It has relatively high selectivity, while when W approaches 0.7 the following XPath expression is no longer uncommon:

/*/*/*/distributor/*/*.

This XPath expression imposes only one concrete filtering condition (i.e., the tag "distributor") and, consequently, matches many incoming XML documents. Hence, such XPath expressions have a lower workload selectivity. The low selectivity property translates to an increased number of matched XPath expressions, which in turn, increases the matching computation time for all algorithms. Even, in face of increased computation time, GPX-ap remains the dominant algorithm, and improves over the next best algorithm (GPX) by at least two times.

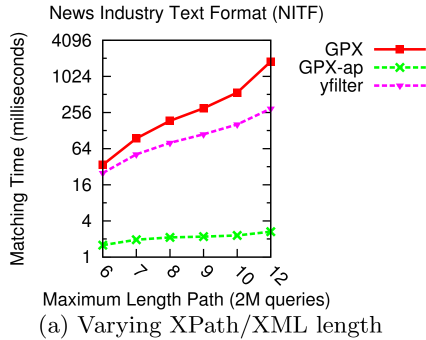
A similar trend is also observed when varying the descendant operator probability, but with a few key distinctions. First, the selectivity does not increase as rapidly as it is increased when the wildcard probability was at 0.7. For example, as DO approaches 0.7, a common XPath expression is:

//body.head//distributor//org//alt-code.

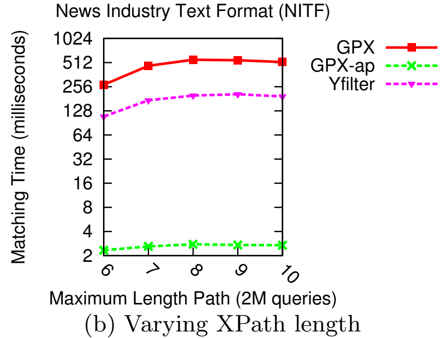
It is rather selective. However, if a correct access predicate is not chosen, then the above expression could partially match many incoming messages due to the nature of the descendant operator, which is translated into the low selective Boolean predicate of *greater than* as oppose to a more selective predicate having *equality* as an operator. Thus, for algorithms that do not incorporate pruning using selectivity (i.e., GPX and YFilter), the matching time increases significantly as the probability of DO increases. However, with the highly effective pruning strategy of GPX-ap, the matching computation time remains virtually unaffected as DO increases.

In general, as the probability of W and DO increases, YFilter suffers due to an increase in the non-determinism of the NFA. Furthermore, the increase in DO also leads to an increase in the size of the NFA. Moreover, the descendant operator introduces states with self-loops, which also increases the number of touched states during filtering. Our unique encoding together with our novel data structures avoid these shortcomings.

Effects of XML/XPath length Another important workload characteristic is the XPath expression length. Therefore, we study the increase in the maximum length of the XPath queries and the maximum number of levels of the



(a) Varying XPath/XML length



(b) Varying XPath length

Figure 5: Varying the XPath/XML length

XML documents for NITF (the PSD DTD is limited to 8 levels) ranging from 6-12. The results are shown in Figure 5a. As expected, the performance of GPX degrades due to the increase in the number of attribute-value pairs resulting from the XML documents which results in an increase in the number of partial matches. However, our GPX-ap gracefully scales as the length of the XPath/XML artifacts increase due to GPX-ap's powerful pruning capability that significantly reduces the number of partial matches. YFilter is also affected as expression and document lengths increase due to the larger NFA size and increased number of states. Similarly, the same trend is observed when the length of XML document was fixed at 10 and the XPath expression length was varied from 6-10 (cf. Figure 5b). In short, GPX-ap is an order of magnitude faster than other approaches.

Value-based predicate filters: Finally, we study the performance of our algorithm when the XPath queries have attribute-based filters. For these experiments, we modified the query generator to output filters on tag names that contain as operator one of the following: $=, \geq, \leq$. We extended GPX-ap to support predicates corresponding to attribute-based filters. All the attributes in the XML documents are evaluated against these predicates. For YFilter, we implemented the selection postponed approach [6]. Note that our approach can be looked at as an inline approach since the predicates are evaluated regardless of whether the queries are structurally matched or not. Next, we investigate the trade-offs between our inline approach and the YFilter selection postponed one.

For the first set of experiments, we created workloads with the same number of queries (half a million), and we varied the number of filters per query ranging from 0 to 10 (see Figures 6(a)-(b) for the results.) For the second set of experiments, we generate workloads with one or respectively two filters per path and varied the number of queries. Figure 6(c) shows the results for the NITF DTD. The results for the PSD DTD are also shown in Figure 6(d).

YFilter is slightly less sensitive when increasing the number of filters per query since the filters are checked only if the queries are structurally matched (selection postponed.) Therefore, when no filter is used GPX-ap is faster by orders of magnitude, and as the number of filters per query increases and approaches 10 filters, GPX-ap suffers some performance degradation, but still improves over YFilter by at least 50% (see Figures 6(a)-(b).)

On the contrary, if most of the queries are structurally matched, YFilter takes a significantly longer time to process queries with only one to two filters per query. This is the case for the PSD DTD (Figure 6(d)) in which the queries are very simple structurally, so most of them are matched in structure, but not in terms of the filters specified. Even for a workload with higher selectivity such as the NIFT DTD (see Figure 6(c)), again, with only one or two filters per query, GPX-ap significantly outperforms YFilter.

6. CONCLUSIONS

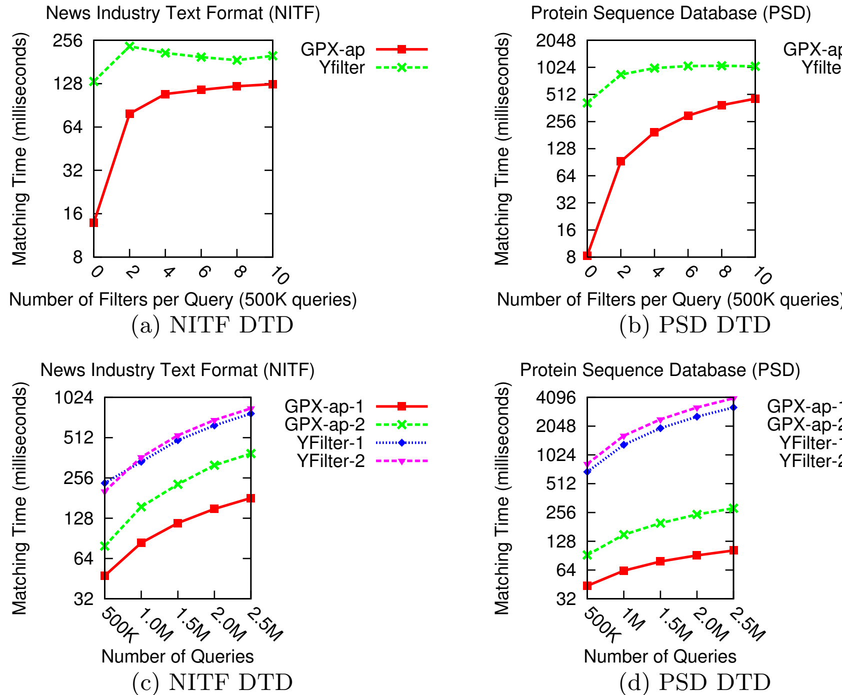
While most existing XML/XPath matching approaches use standard techniques such as NFAs or DFAs for processing XPath expressions, we altogether diverted and proposed a fundamentally new approach for XML/XPath matching. Our novel proposal is GPX-Matcher, a matching algorithm that determines interested subscribers in a content-based publish/subscribe system for which subscriptions are captured using XPath expressions while publications are captured as XML documents. The key idea behind our algorithm is that XML messages are translated into sets of attribute-value pairs that are evaluated over conjunctions of Boolean predicates resulting from our unique XPath expression encoding. Our extensive evaluations illustrate the scalability of GPX-Matcher for XML/XPath matching involving millions of XPath expressions. In our experiments, we show that GPX-Matcher gracefully scales for workloads of up to four million subscriptions. On workloads without duplicate subscriptions exhibiting high selectivity (i.e., few subscriptions match the incoming message), GPX-Matcher clearly outperforms state-of-the-art techniques by orders of magnitude. Moreover, for workloads including duplicates and high matching load, GPX-Matcher also consistently outperforms alternatives due to its unique encoding and novel data structures that effectively exploit workload selectivity. Lastly, GPX-Matcher is highly robust while changing key workload parameters such as wildcard probability, descendant operator probability, XML/XPath depth, and number of filters per XPath expression.

7. REFERENCES

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *ACM PODC'99*.

[2] G. Ashayer, H. Leung, and H.-A. Jacobsen. Predicate matching and subscription matching in publish/subscribe systems. In *DEBS'02*.

[3] N. Bruno, L. Gravano, N. Koudas, and D. Srivastava. Navigation-vs. Index-based XML multi-query



(a) NITF DTD (b) PSD DTD

(c) NITF DTD (d) PSD DTD

Figure 6: Attribute-based filters

processing. In *ICDE'03*.

[4] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal. AFilter: adaptable XML filtering with prefix-caching suffix-clustering. In *VLDB'06*.

[5] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *ICDE*, 2002.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS'03*.

[7] F. Fabret, H.-A. Jacobesen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD'01*.

[8] E. Fidler, H.-A. Jacobsen, G. Li, and S. Mankovski. The PADRES distributed publish/subscribe system. In *ICFI'05*.

[9] P. M. Fischer and D. Kossmann. Batched processing for information filters. In *ICDE'05*.

[10] K. J. Gough and G. Smith. Efficient recognition of events in distributed systems. In *ACSC18'95*.

[11] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *ICDT'02*.

[12] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *SIGMOD'03*.

[13] S. Hou and H.-A. Jacobsen. Predicate-based filtering of XPath expressions. In *ICDE'06*.

[14] R. S. Kazemzadeh and H.-A. Jacobsen. Reliable and highly available distributed publish/subscribe service. In *SRDS'09*.

[15] J. Kwon, P. Rao, B. Moon, and S. Lee. Fast XML document filtering by sequencing twig patterns. *ACM TOIT'09*.

[16] J. Kwon, P. Rao, B. Moon, and S. Lee. Value-based predicate filtering of XML documents. *DKE'08*.

[17] G. Li, S. Hou, and H.-A. Jacobsen. Routing of XML and XPath queries in data dissemination networks. In *ICDCS'08*.

[18] G. Li, S. Hou, and H.-A. Jacobsen. A unified approach to routing, covering and merging in publish/subscribe systems based on modified binary decision diagrams. In *ICDCS'05*.

[19] G. Li, S. Hou, and H.-A. Jacobsen. XML routing in data dissemination networks. In *ICDE'07*.

[20] G. Li, V. Muthusamy, and H.-A. Jacobsen. A distributed service-oriented architecture for business process execution. *ACM TWEB'10*.

[21] M. M. Moro, P. Bakalov, and V. J. Tsotras. Early profile pruning on XML-aware publish-subscribe systems. In *VLDB'07*.

[22] News Industry Text Format (NIFT). http://www.nitf.org.

[23] J. Pereira, F. Fabret, H.-A. Jacobsen, F. Llirbat, and D. Shasha. WebFilter: A high-throughput XML-based publish and subscribe system. VLDB '01.

[24] PIR-International Protein Sequence Database (PSD). http://pir.georgetown.edu.

[25] M. Sadoghi, M. Labrecque, H. Singh, S. Warren, and H.-A. Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. In *VLDB'10*.

[26] S. E. Whang, H. Garcia-Molina, C. Brower, J. Shanmugasundaram, S. Vassilvitskii, E. Vee, and R. Yerneni. Indexing boolean expressions. *VLDB'09*.

[27] T. W. Yan and H. García-Molina. Index structures for selective dissemination of information under the boolean model. *ACM Trans. Database Syst.'94*.