Predicate-based Filtering of XPath Expressions

Shuang Hou H.-A. Jacobsen

Technical Report CSRG-514
Department of Computer Science, University of Toronto

Abstract

The XML/XPath filtering problem has found wide-spread interest. In this paper, we propose a novel algorithm for solving it. Our approach encodes XPath expressions (XPEs) as ordered sets of predicates and translates XML documents into sets of tuples, which are evaluated over these predicates. Predicates representing overlapping portions of XPEs are stored and processed once, thus fully exploiting potential overlap in XPEs. We experimentally evaluate the performance of our algorithm, demonstrating its scalability to millions of XPEs, with matching performance in the millisecond range. We show interesting trade-offs to alternative approaches.

1 Introduction

XML has become the lingua franca on the Internet, with applications ranging from life sciences [16] to news [15] and advertisement dissemination, among many others. Moreover, XML is changing the way applications exchange messages, the way systems are integrated, and the way information is stored and processed on the Internet [19]. Thus, it is not surprising that XML has been widely advocated as data representation format of choice for selective information dissemination applications and for content-based message routing [8, 5, 13, 4]. In these scenarios, it is expected that the filtering engines have to be capable of processing several millions of filter expressions over high XML document input rates.

For selective information dissemination applications, an XPath filter expression designates an entity's (user or system) interest to receive information of the specified nature. The XML document constitutes the content to be selectively disseminated. XPath expressions [20] have been widely proposed for modeling such filters [2, 8, 5, 13, 12, 4, 6]. An XPath expression (XPE) can specify filter constraints on the document's structure, its attributes, and its content. Filter selectivity on a very fine-grained level is thus possible.

The XML/XPath *filtering problem* determines the set of matching XPEs among all expressions to be processed for a given input XML document. It is this problem that we set out to solve with a novel filtering algorithm. The algorithm we propose is fundamentally different from existing approaches [2, 8, 5, 13, 12, 4, 6], which are based on finite state machines, nodeterministic finite state machines, push-down automaton, trie-based indices, or relational database querying techniques. The major challenge of this filtering problem is to efficiently handle a large number of XPEs. Our goal and main difference to related approaches is to exploit the overlap among XPEs as much as possible. Our techniques go beyond exclusively considering prefix overlap. The main idea of our algorithm is to translate XPEs into ordered sets of predicates, such that the common parts among XPEs will be stored and processed only once, and the partial matching result will be shared among all expressions that contain these common parts. For example, in the two XPEs $s_1 : a/b/c/d$ and $s_2 : b//b/c$, we translate the overlapping parts, b/c, into one predicate. This predicate will be evaluated against the XML document only once. It is also shared by other expressions that contain it.

The first contribution of this paper is a completely novel algorithm for solving the XML filtering problem. In a second contribution, we propose several refinements and optimizations for our approach, such as the predicate index, the prefix covering index, the access predicate scheme, and extensions for processing nested path and attribute-based filters. Third, we perform a thorough experimental analysis that evaluates the performance of our algorithm under varying experimental conditions. We demonstrate the scalability of the algorithm to millions of XPEs, and present a detailed trade-off analysis against two alternative techniques from different categories of the solution space of the filtering problem.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents our XPE encoding and XML encoding schemes. Section 4 presents our matching algorithm. Section 5 demonstrates the handling of path filters, both attribute-based filters and nested path filters. Section 6 presents a thorough experimental analysis and comparison to alternative approaches.

2 Related Work

XML filtering and XML query processing have been widely studied. Much work exists on storing, retrieving and processing XML data through relational data management approaches [18, 11, 17]. Due to space limitations, we focus our discussion on the work that is most closely related to our approach, namely alternative approaches for XML/XPath filtering.

These alternative approaches can be broadly classified into automaton-based approaches and index-based approaches. Automaton-based approaches [2, 8, 12, 13] build an automaton based on the XPEs processed by the system. The transitions in the automaton are triggered by the tags of the XML document path being processed. XFilter [2] treats each XPE as a finite state machine. This approach is not able to adequately handle overlap, especially, prefix overlap between expressions. YFilter [9] extends XFilter by building a non-deterministic finite automaton (NFA) for all XPEs in the system. It honors common prefixes among expressions. Each final state of the NFA is associated with a list of expressions. The parsing of the XML document, 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 XML document processed. The difference between a traditional NFA and the YFilter data structure is that the execution of the YFilter NFA does not stop when the first final state is reached, but continues until all possible accepting states have been visited. Thus, the algorithm can determine all matching expressions stored in the NFA. The YFilter [9] paper demonstrates superior performance to XTrie [5], which is the reason we chose YFilter for our performance comparisons. Another automaton-based approach, XPush [13], lazily constructs a single deterministic pushdown automaton for XPEs in the system. A drawback of the approach for online selective information processing is the difficulty in adding or deleting queries from the automaton constructed.

The index-based algorithms [4, 5, 14] take advantage of precomputed schemes on either the XML documents or the XPath expressions. XTrie [5] proposes a trie-based index structure, which decomposes the XPEs to substrings that only contain parent-child operators. As a result, the processing of these common substrings among queries can be shared. Despite this sharing, YFilter [9] has been demonstrated to have better performance on certain workloads. Lakshman *et al.* [14] propose an index structure that manages XPEs for solving the filtering problem. The algorithms requires two passes over an XML document processed. Index-Filter [4] addresses the problem of obtaining *all matches* for each expression stored in the system. It answers multiple XML path queries by building indexes over the XML document elements either in a subcomputation stage or on the fly, which avoids processing large portions of the input document that are guaranteed not to be part of any match. The algorithm takes advantage of a prefix tree representation of the set of XML queries to share the computation during multiple evaluations. It is efficient especially in the case of large XML documents or set of documents. We have chosen Index-Filter in our performance comparison as a representative of index-based filtering schemes for solving the matching problem.

3 XPath and XML Encoding

In this section, we motivate and describe the encoding of XPEs with a predicate calculus and the encoding of XML documents as sets of tuples. Our algorithm, described in the following section, efficiently evaluates the tuples over the predicates to determine the matching XPEs.

3.1 Basic Idea

The basic idea underlying our encoding is to record the position information of each tag and the relative position information for each two adjacent tags in the XPEs and XML documents, respectively. From this we can determine whether the position information recorded for each XPE is matched by the position information represented in the current XML document.

Each XPE is translated into an *ordered set of predicates*, where each predicate is an (attribute, operator, value)–triple that expresses a constraint over the value of the attribute (i.e., the position of the tag). More formally, an XPE is described as $s : (a_1, o_1, v_1) \mapsto (a_2, o_2, v_2) \mapsto ... \mapsto (a_n, o_n, v_n)$, where a_i represents either a tag name variable or a pair of tag name variables, o_i represents a relational operator and "$\mapsto$" represents the sequence of predicates, that is, $(a_{i-1}, o_{i-1}, v_{i-1})$ should be ahead of (a_i, o_i, v_i). For example, $a/*/b//c$ is translated to $(d(p_a, p_b), =, 2) \mapsto (d(p_b, p_c), \geq, 1)$, where each predicate encodes the relative position information of each two adjacent tags (the predicates will be discussed shortly.) The order-preserving requirement imposed by the sequencing of predicates is an essential part of our approach and will be clarified in the sequel.

In our approach, we use the common interpretation of an XML document as a tree of nodes and consider each path from the root node in this tree to a leaf node, separately.[1] These paths are "collected" in the parsing stage. In our implementation we use a SAX parser and extract one path at a time from the document. This path extraction adds a negligible amount of overhead, as we illustrate in our experiments. Thus, we decompose each XML document into a set of XML paths and each path is encoded as a set of (attribute-value)-pairs. where each pair is, either a single XML tag name, a_i, and its position value, v_i, in the document path, or a special attribute $length$ representing the length of this XML path. For example, the XML path, $e = (t_1, t_2, ..., t_n)$, in our encoding is $\{(length, n), (t_1, 1), (t_2, 2), ..., (t_n, n)\}$. Further examples will follow in Section 3.3.

For each XML document, we evaluate all its XML paths against all XPEs in the system. An XPE is matched by a given XML document *iff*[2] the evaluation of the XPE over the XML document is a non-empty set of nodes in the XML tree. Our data structure and algorithm exploit overlap in XPEs by storing and evaluating unique predicates only.

If any XPE, s_i, in the system is matched by any document path of an incoming XML document, $D = \{e_1, e_2, ..., e_n\}$, we say the XPE is matched by this document. The objective of this matching semantic is to filter out XML documents matching any XPE stored in the system. Note that the XPEs, s_i, are single-path expressions in the following discussion. We will discuss the matching of nested path expressions in Section 5.

3.2 XPath Expression Encoding

First, we present our predicate language for encoding XPE operators. In this section, we focus on parent-child operator (/), wildcard operator (*), and ancestor-descendant operator (//). We then show how to represent entire XPEs in this language. We will discuss how to support attribute-based and nested path filters with our predicate language in Section 5. This extensibility demonstrates the principle for adding other XPath language support to our approach.

Predicate Language: We use an extensible predicate language to express constraints over the absolute and the relative position of tags in the XPEs. We define the following four types of predicates to represent absolute XPEs, relative XPEs, wildcards in XPEs, and //-operators. Later, we show how to extend this language to process attribute filters in XPEs and nested path filters.

Absolute predicates: The predicate, (p_t, op, v), represents a constraint on the position of the tag, t, in the XPE. The operator, op, can be either $=$ or $\geq$. For the equality operator, this predicate represents the absolute position of a tag in the absolute XPE (i.e., an XPE explicitly starting at the root element and not including descendant operators). For example, in the XPE, $/*/t_1$, the predicate, $(p_{t_1}, =, 2)$, is used to represent the absolute position information of tag t_1 in s. For the greater-than-or-equal operator, this predicate represents the tag, which appears after some descendant operator in the absolute XPE, or represents the first tag in the relative XPE, which is not a wildcard. For example, in the absolute XPE, $/*//t_1$, $(p_{t_1}, \geq, 2)$ is used to represent the position information of tag t_1. In the relative XPE, $*/*/t_1$, the predicate, $(p_{t_1}, \geq, 3)$ is used to represent the position information of tag t_1.

Relative predicates: The predicate, $(d(p_{t_1}, p_{t_2}), op, v)$, represents a constraint between the relative position of the tag, t_2, with respect to the tag, t_1. The operator op can be either $=$ or $\geq$. For the equality operator, this predicate represents tags in XPEs that are not linked by descendant operators. For example, for $t_1/*/t_2$, the predicate, $(d(p_{t_1}, p_{t_2}), =, 2)$ is used to represent the relative position information between the tag t_1 and t_2. Note that the order between these tags in the path is important. The predicate $(d(p_{t_1}, p_{t_2}), =, 2)$ indicates that the tag t_1 should appear two location steps before the tag t_2 in the XML path. For the greater-than-or-equal operator, this predicate represents tags that are linked by descendant operators. For example, in the XPE $t_1//t_2$, the predicate, $(d(p_{t_1}, p_{t_2}), \geq, 1)$ expresses that the tag, t_1, must appear more than one location step before the

[1] For processing nested path filters, we need to keep state between processing different paths, as we illustrate in Section 5.

[2] We will use the notation *iff* to mean "if and only if" in this paper.

tag, t_2, in any matching document path.

End-of-path predicates: 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. It is for those tags in the absolute or relative XPE, which are followed by wildcards only. For example, for $/t_1/*/*$, not only do we need the predicate, $(p_{t_1}, =, 1)$, to represent the position constraint of tag, t_1, we also need the predicate, $(p_{t_1}^{\dashv}, \geq, 2)$, to represent the position constraint of this tag relative to the end of the XML path. There must be two or more tags following t_1 in a matching XML path to satisfy the expression.

Length-of-expression predicates: The predicate, $(length, \geq, v)$, is a special predicate, which does not refer to any tag names and their positions, but to the length of the XPE. It represents a constraint on the length of the XML path. It is for those XPEs that contain only wildcards. For example, the XPE, $s_1 : /*/*/*$, is translated into $(length, \geq, 3)$, which indicates that all XML paths containing enough tags will match this XPE. Note that the expression $s_2 : */*/*$ has the same mapping, since both of them actually require that the length of the XML path is at least 3. We do not distinguish these two kinds of expressions in this paper, since no distinction is required to satisfy the matching semantic we set out to solve (i.e., any document path of length at least 3 will correctly match both expressions.)

Mapping XPEs to Predicates: This section illustrates, first, through various examples and then generically how different XPEs are translated to the ordered sets of predicates of the types defined in the previous section. These predicates record the position information of the first non-wildcarded location step and the relative position information between every two adjacent tags. The objective is to record just enough information to uniquely represent each XPE. The mapping also honors overlap between different XPEs and represents overlapping parts of expressions with the same predicates. Our algorithm only stores and processes unique predicates once, thus leveraging the overlap in expressions.

Simple XPEs: A simple XPE only contains parent-child operators. It is represented with absolute and relative predicates as illustrated below.

XPE	**Ordered predicates**
$s_1 : /a/b/b$	$(p_a, =, 1) \mapsto (d(p_a, p_b), =, 1) \mapsto (d(p_b, p_b), =, 1)$
$s_2 : a$	$(p_a, \geq, 1)$
$s_3 : a/a/b/c$	$(d(p_a, p_a), =, 1) \mapsto (d(p_a, p_b), =, 1) \mapsto (d(p_b, p_c), =, 1)$

For s_3, we do not need the predicate $(p_a, \geq, 1)$ for the first tag, a, since other predicates already indicate that the constraint on the position of the first tag should be at least 1. Note, for certain XPEs alternative encodings come to mind. However, it is these encodings here for which we can prove the correctness of encodings and matching algorithm in Appendix A.

The *general absolute XPE,* $/t_1/t_2/.../t_{n-1}/t_n$, is encoded as $(p_{t_1}, =, 1) \mapsto (d(p_{t_1}, p_{t_2}), =, 1) \mapsto ... \mapsto (d(p_{t_{n-1}}, p_{t_n}), =, 1)$. The *general relative XPE,* $t_1/t_2/.../t_{n-1}/t_n$, is encoded as $(d(p_{t_1}, p_{t_2}), =, 1) \mapsto ... \mapsto (d(p_{t_{n-1}}, p_{t_n}), =, 1)$.

Wildcards in XPEs: Wildcards can appear at the beginning, the middle and the end of an XPE. In our encoding of XPEs with wildcards, we simply bypass the wildcarded location steps as illustrated below.

XPE	**Ordered predicates**
$s_4 : /a/*/*/b$	$(p_a, =, 1) \mapsto (d(p_a, p_b), =, 3)$
$s_5 : /a/b/*/*$	$(p_a, =, 1) \mapsto (d(p_a, p_b), =, 1) \mapsto (p_b^{\dashv}, \geq, 2)$
$s_6 : /*/a/b$	$(p_a, =, 2) \mapsto (d(p_a, p_b), =, 1)$
$s_7 : /*/*/*/*$	$(length, \geq, 4)$
$s_8 : a/b/*/*$	$(d(p_a, p_b), =, 1) \mapsto (p_b^{\dashv}, \geq, 2)$
$s_9 : */*/a/*/b$	$(p_a, \geq, 3) \mapsto (d(p_a, p_b), =, 2)$
$s_{10} : a/*/*/b/c$	$(d(p_a, p_b), =, 3) \mapsto (d(p_b, p_c), =, 1)$
$s_{11} : */*/*/*$	$(length, \geq, 4)$

In the expression, s_9, we need the predicate $(p_a, \geq, 3)$ to represent the position constraint of tag, a, since the other predicates do not represent any position information for the first non-wildcarded location step.

The *general absolute XPE,* $/*/.../*/t_1/.../t_i/*/.../*/t_{i+1}/.../t_n/*/.../*$, is encoded as $(p_{t_1}, =, m_1) \mapsto (d(p_{t_1}, p_{t_2}), =, 1) \mapsto ... \mapsto (d(p_{t_{i-1}}, p_{t_i}), =, 1) \mapsto (d(p_{t_i}, p_{t_{i+1}}), =, m_2) \mapsto ... \mapsto (d(p_{t_{n-1}}, p_{t_n}), =, 1) \mapsto (p_{t_n}^{\dashv}, \geq, m_3)$, where m_1, m_2 and m_3 are consecutive wildcards appearing at the beginning, middle and end of the XPE, respectively. The mapping for a *general relative XPE* with wildcards is similar, except that the first predicate is replaced by the predicate, $(p_{t_1}, \geq, m_1)$.

Descendant operator in XPEs: Descendant operators indicate that there are more than one location step between two tags as illustrated below.

XPE	**Ordered predicates**
$s_{12} : /a//b/c$	$(p_a, =, 1) \mapsto (d(p_a, p_b), \geq, 1) \mapsto (d(p_b, p_c), =, 1)$
$s_{13} : /*/b//c/*$	$(p_b, =, 2) \mapsto (d(p_b, p_c), \geq, 1) \mapsto (p_c^{\dashv}, \geq, 1)$
$s_{14} : a/b//c$	$(d(p_a, p_b), =, 1) \mapsto (d(p_b, p_c), \geq, 1)$
$s_{15} : */a/*/b//c/*/*$	$(p_a, \geq, 2) \mapsto (d(p_a, p_b), =, 2) \mapsto (d(p_b, p_c), \geq, 1) \mapsto (p_c^{\dashv}, \geq, 2)$

The *general form of an absolute XPE* with descendant operator(s), $/t_1/.../t_i//t_{i+1}/.../t_n$, is encoded as $(p_{t_1}, =, 1) \mapsto (d(p_{t_1}, p_{t_2}), =, 1) \mapsto ... \mapsto (d(p_{t_{i-1}}, p_{t_i}), =, 1) \mapsto (d(p_{t_i}, p_{t_{i+1}}), \geq, 1) \mapsto ... \mapsto (d(p_{t_{n-1}}, p_{t_n}), =, 1)$, where the descendant operator can appear at any location step of the XPE one or more than one time. The encoding for the *general relative XPE* with descendant operators is the same, except for missing first predicate.

As a result, the common part among the XPEs are mapped to the same one or several predicates, which will be stored and evaluated only once during the matching. For example, a/b is translated into only one predicate $(d(p_a, p_b), =, 1)$ in the above examples in spite of the position it appears in the XPEs.

It is important to note that our mapping is not predicate-order invariant. That is, the order of predicates can not be changed because different tags may have the same name, and "$\mapsto$" determines the sequence of predicates appearing in the mapping, which is the reason for the use of an ordered set, rather than a traditional set in our formalization. For example, $(d(p_a, p_c), =, 1) \mapsto (d(p_c, p_a), =, 2) \mapsto (d(p_a, p_c), \geq, 1)$ is used to represent the XPE, $s_1 : a/c/*/a//c$, where the first tag, a, and the third tag, a, are different tags but have the same name. We identify them by looking at $(d(p_a, p_c), =, 1)$ and $(d(p_a, p_c), \geq, 1)$, which indicate that the first tag, a, is followed by tag, $/c$, and another tag, a, is followed by $//c$. If we change the order of these two predicates, say $(d(p_a, p_c), \geq, 1) \mapsto (d(p_c, p_a), =, 2) \mapsto (d(p_a, p_c), =, 1)$, another XPE, $s_2 : a//c/*/a/c$, is represented. That is, we use the order of these predicates to represent the sequence of tags appearing in the expression.

3.3 XML Document Encoding

An XML document, D, consists of a number of document paths, $D = (e_1, e_2, ..., e_n)$. Each document path, $e_i = (t_1, t_2, ..., t_n)$, consists of a sequence of tag names, t_i, optional attributes and corresponding values, and the content associated with each tag. For simplicity we only refer to the tag names in the above formalization.

The intuition behind our document path encoding is that we translate each XML document path to a set of (attribute, value)–pairs, which are evaluated against the XPE predicates.

Our encoding records the information of the path length and the position of each tag in the document path. We translate each XML document path, $e = (t_1, t_2, ..., t_n)$, to the following set of tuples:

$$(length, n), (t_1, 1), (t_2, 2), ..., (t_n, n)$$

where $(length, n)$ represents the length of the XML path, which is required to evaluate the predicates $(p_t^{\dashv}, \geq, v)$ and $(length, \geq, v)$. The pair (t_i, i) determines that the position of tag t_i is i in this path. This is required to evaluate the predicates (p_{t_i}, op, v) and $(p_{t_i}^{\dashv}, \geq, v)$. In order to evaluate the predicate $(d(p_{t_i}, p_{t_j}), op, v)$, two pairs, (t_i, i) and (t_j, j), are required at the same time to determine the distance between tag t_i and t_j. The interpretation as (attribute, value)–pairs derives from the fact that "length" and the tag name are attribute designators. The range of the value is $[1, n]$ (n is the length of the XML path). We refer to this set of (attribute, value)–pairs for each document path as *publication*. Based on this encoding, all predicates are evaluated against each tuple or a pair of tuples in the publication according to a set of predefined rules for predicate matching.

In our encoding, we distinguish the case of different tags with the same name occurring in the same XML document path. Each tag is annotated with an *occurrence number*, for that purpose, which counts the number of times this tag was already present in this path.

Example 1: Consider the path, $e = (a, b, c, a, b, c)$, which is annotated with superscripted occurrence numbers, $e = (a^1, b^1, c^1, a^2, b^2, c^2)$. It is translated into the following publication:

$$(length, 6), (a^1, 1), (b^1, 2), (c^1, 3), (a^2, 4), (b^2, 5), (c^2, 6)$$

where $(a^2, 4)$ indicates that the second occurring a in this document path is at position 4.
This encoding is built in the XML document parsing stage and does not require additional processing, except for collecting the occurrence numbers for each tag appearing in the current path. In our implementation this is based on a hash-table that keeps track of the already for the current path seen tag names.

4 Matching Algorithm

In this section, we discuss the matching algorithm. It has two stages. First, the predicate matching stage determines all matching predicates and, second, the XPath expression matching stage. The latter stage computes the matching XPEs based on the matching predicate information determined in the first stage. We discuss both stages in turn. The interested reader is referred to Appendix A for proving the correctness of our matching algorithm based on above encoding.

4.1 Predicate Matching

4.1.1 Predicate Evaluation

Each of the four predicates is evaluated over a document path according to the following rules.

Absolute predicates: The predicate, (p_t, op, v), is matched by a given tuple *iff* the tuple is (t, v'), and it satisfies the relation $v'\ op\ v$.

Relative predicates: The predicate, $(d(p_{t_1}, p_{t_2}), op, v)$, is matched by a given pair of tuples *iff* the tuples are (t_1, v_1), (t_2, v_2), and they satisfy the relation $(v_2 - v_1)\ op\ v$. For example, given tuples $(a, 2)$ and $(b, 6)$, $(d(p_a, p_b), =, 2)$ is not matched since $6 - 2 = 2$ does not hold.

End-of-path predicates: The predicate, $(p_t^{\dashv}, \geq, v)$, is matched by a given tuple *iff* the tuple is (t, v'), and it satisfies the relation $l - v' \geq v$ (l is the length value).

Length-of-expression predicates: The predicate, $(length, \geq, v)$, is matched by a given tuple *iff* the tuple is $(length, v')$, and it satisfies the relation $v' \geq v$.

For each tuple in a publication (i.e., document path), we record the matching predicates and the occurrence number of the tag associated with each match. Both are used to determine the matching XPEs in the second stage of the algorithm.

Table 1. Predicate Matching Result

XPE	Predicates	Matching Results
$a//b/c$	$(d(p_a, p_b), \geq, 1)$	$(\mathbf{a^1}, \mathbf{b^1}), (a^1, b^2), (a^2, b^2)$
	$(d(p_b, p_c), =, 1)$	$(\mathbf{b^1}, \mathbf{c^1}), (b^2, c^2)$
$c//b//a$	$(d(p_c, p_b), \geq, 1)$	(c^1, b^2)
	$(d(p_b, p_a), \geq, 1)$	(b^1, a^2)

Example 2: Considering the XML path from Example 1 and the two XPEs: $a//b/c$ and $c//b//a$, the individual predicate matching results are shown in Table 1. For expression $a//b/c$, which translates to $(d(p_a, p_b), \geq, 1) \mapsto (d(p_b, p_c), =, 1)$, each predicate with its corresponding matching results is shown. The combination of two boldface matching results is one of the true matches for $a//b/c$. From these results, it can be seen that not all combinations of the predicate matching results constitutes a true XPE match. For example, predicate matching results (a^1, b^2) and (b^1, c^1) are not a true XPE match, since the occurrence numbers (i.e., superscripts) indicate that the first predicate is matched by the second b in the XML path, while the second predicate is matched by the first b in the path. The same situation occurs in the other expression, which indicates that there is no match for $c//b//a$. The *occurrence determination* algorithm, described in Section 4.2.1, will determine whether at least one true match exists, among the individual predicate matching results, or not for each XPE.

4.1.2 Predicate Index

We maintain a predicate index that manages distinct predicates through multiple stages of hashtables. Suppose that we have two XPEs, $/a/*/c$ and $*/a/*/c/*/*/*$, the common part $a/*/c$ is represented as $(d(p_a, p_c), =, 2)$ and only stored once in our predicate index, as shown in Figure 1. All predicates are managed in the first stage of the index according to their type. For absolute predicates, i.e., (p_t, op, v), we form hash keys by using the tag name to access a separate array per operator that stores predicate identifiers (pids). Each array is indexed by the predicate's value. The length of the array depends on the maximum length of the XPEs supported by the system. For relative predicates, i.e., $(d(p_{t_1}, p_{t_2}), op, v)$, we form a first-level hash key with the first tag and a second-level hash key with the second tag. The first-level hashtable indexes into a second-level table, which indexes into the same structure as the table for managing absolute predicates. For end-of-path predicates, i.e., $(p_t^{-1}, \geq, v)$, we form a hash key based on the tag and manage one array per tag that keeps track of all predicates of this type. For length-of-path predicates, i.e., $(length, \geq, v)$, we only keep an array and do not need additional keys since $length$ and $\geq$ are fixed.

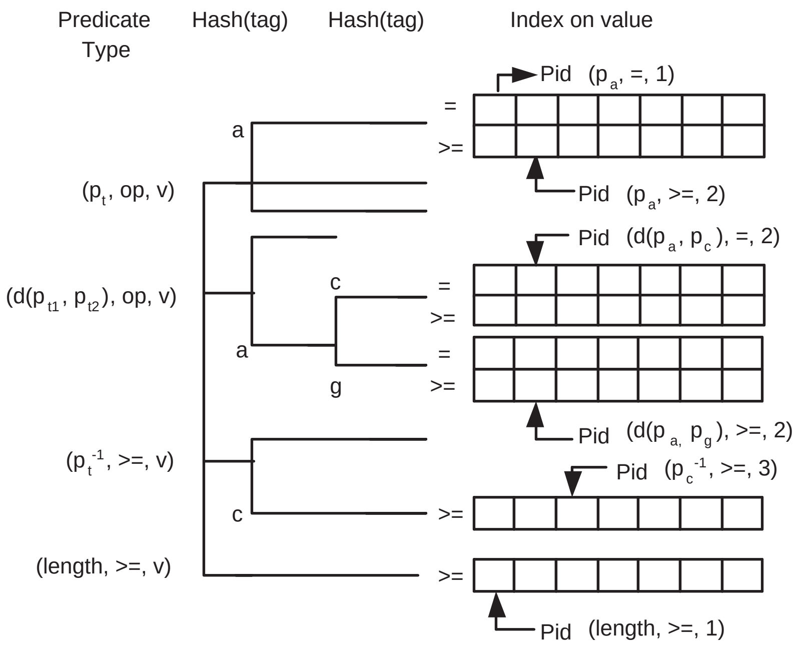


Figure 1. Predicate Index

A predicate is inserted into the index by hashing on tag name(s) and indexing into the corresponding array with the predicate value. If a pid is found, it is returned, indicating that the exact same predicate is already stored in the system, otherwise a new pid is created for this predicate and stored at the indexed location in the array.

Based on predicate matching rules defined in Section 4.1.1, the predicate matching is straightforward by using the predicate index. For absolute predicates, given each tuple in the document path, a matching predicate is identified by checking for existing pids in the associated array at the index position identified by the value in this tuple (for equality predicates) and at all index positions in the array larger than this value (for greater-than-or-equal predicates). For end-of-path and length-of-expression predicates, the evaluation is similar except that the length of document path is applied. For relative predicates, the evaluation must correlate a pair of tuples and the index position is identified by the difference of the positions of the second-level and first-level tags in the document path.

4.2 XPath Expression Matching

The previous predicate matching stage results in a set of matched predicates. For each predicate we record matching occurrences, i.e., an occurrence number of the tag that matches the predicate. From this information we compute the matching XPEs. In this section, we first present the algorithm that, based on the matching occurrences of predicates of an XPE, determines whether the expression is matched. We then explore two optimized organizations of XPE predicates to more efficiently identify matching candidates.

4.2.1 Determining a Matching Expression

This section describes how to determine a matching XPE from the matching occurrence information of its constituent predicates. An XPE, s, is the ordered set of predicates, $s = pre_1 \mapsto pre_2 \mapsto ... \mapsto pre_n$. The predicate matching result for the predicate over one tag and two tags is the set of occurrence numbers and occurrence number pairs, respectively. In our notation (and implementation) we omit the tag names and only keep track of occurrence numbers, as tag information is no longer needed. For example, in Table 1, the predicate matching occurrence results for expression, $a//b/c$ are $\{(1,1),(1,2),(2,2)\} \mapsto \{(1,1),(2,2)\}$. Here, the first tuple $(1,1)$ specifies that the first predicate of $a//b/c$ is matched by the first tag, a, and the first tag, b, in the document path. The second tuple $(1,1)$ specifies that the second predicate of $a//b/c$ is matched by the first tag, b, and the first tag, c, in the XML document path. Not each combination of matching occurrence results of predicates are a true match (see Example 2). The combination is a true match if and only if the second occurrence number of pre_{i-1} is the same as the first occurrence number of pre_i $(2 \leq i \leq n)$. A correct combination has the following format: $(o_1^1, o_2^1), (o_1^2, o_2^2), ..., (o_1^n, o_2^n)$, where (o_1^i, o_2^i) is selected from the predicate matching results of pre_i, and $o_2^{i-1} = o_1^i$ holds for each i $(2 \leq i \leq n)$.

Depending on the predicate type, a matching occurrence result has one or two values, i.e., one per tag name variable in the predicate. In our notation, to simplify the discussion, we duplicate the occurrence result, if only one value is present. For relative predicates, two different values are required and cannot be avoided (i.e., one per matching tag name variable in the document path.)

The problem of finding a sequence of occurrence numbers is a constraint satisfaction problem [7], where the constraints are $o_2^{i-1} = o_1^i$ for each i $(2 \leq i \leq n)$. We use a backtracking algorithm [7, 3] to solve this constraint satisfaction problem. Our algorithm is shown in Algorithm 1.

Algorithm 1 Occurrence Determination Algorithm

```
Require: the ordered matching results $R = \{R_1, R_2, ..., R_n\}$
Ensure: match or noMatch
 1: $current \leftarrow 1, step \leftarrow 1, back \leftarrow false$
 2: for each $R_i \in R$ do
 3:   if $R_i = \varnothing$ then
 4:     return noMatch
 5:   end if
 6: end for
 7: $R'_1 \leftarrow R_1$, select one pair $(o_1, o_2)$ from $R'_1$ and then delete it from $R'_1$, $p_{current} \leftarrow (o_1, o_2)$
 8: while true do
 9:   if $back = false$ then
10:     if $current = n$ then
11:       return match
12:     else
13:       $current++, step \leftarrow current$, and $R'_{current} \leftarrow R_{current}(o_2)$
14:     end if
15:   end if
16:   if $R'_{current} \neq \varnothing$ then
17:     select one pair $(o_1, o_2)$ from $R'_{current}$, and remove it from $R'_{current}$,$p_{current} \leftarrow (o_1, o_2)$, $back \leftarrow false$
18:   else
19:     $step--$
20:     while $R'_{step} = \varnothing$ and $step \neq 0$ do
21:       $step--$
22:     end while
23:     if $step = 0$ then
24:       return noMatch
25:     else
26:       $current \leftarrow step, back \leftarrow true$
27:     end if
28:   end if
29: end while
```

The input to the algorithm are predicate matching results, $R = \{R_1, R_2, ..., R_n\}$, for the given XPE, in correct order, where R_i is the predicate matching result for pre_i. The algorithm exits with *match* if it finds a correct match, and returns *noMatch* if it finds no matching combination. The algorithm works by traversing the space of partial matching solutions (i.e., sequences of matching occurrence numbers) in a depth-first manner and prunes the search space by ruling out discontinuing occurrences (i.e., occurrences such as $(1,1)$ and $(2,3)$, for instance, which do not constitute matching candidates.) The algorithm stops as a matching sequence is found. Were we to continue the algorithm, all possible matches would be found. To implement the specified matching semantic we only need to determine one match, not all possible matches.

We now discuss the algorithm in detail. The variables, initialized in line 1 are $current$ representing the current predicate, $step$ representing the backtracking depth of the algorithm, and $back$ a flag to record the algorithm's "direction" — forward or backward. The algorithm immediately returns *noMatch* if there is any predicate without matching result among the input (line 2). Next, the algorithm (randomly) selects one pair, (o_1, o_2), from R'_1, removes it, and lets R'_1 be the set of available pairs for pre_1 (line 3). Then the algorithm selects all pairs, whose first occurrence number is o_2, from R_2, and stores these pairs in R'_2. If R'_2 is $\varnothing$, then this indicates that no correct combination could start with (o_1, o_2), therefore, we have to select another occurrence number pair from R'_1. This step continues for all i. That is, select one pair, (o^i_1, o^i_2), from R'_i, which is the current selected pair for pre_i, and removes it from R'_i (line 8). We build R'_{i+1} by selecting pairs, whose first occurrence number is o^i_2, from R_{i+1}. R'_{i+1} is the set of available matching results for pre_{i+1} (line 7). If $R'_{i+1} = \varnothing$, then there is no correct choice for currently selected occurrence numbers, (o^i_1, o^i_2). We, therefore, backtrack to i, chose the next available, (o^i_1, o^i_2), if $R'_i \neq \varnothing$. The algorithm returns *match* as the first (o^n_1, o^n_2) (line 6) is found, and returns *noMatch*, if R'_1 is $\varnothing$ (line 10).

4.2.2 Optimized Expression Organizations

The objective is to apply the occurrence determination algorithm as few times as possible. The algorithm does not have to be applied, if for a given XPE, one or more of its predicates is not satisfied. The algorithm does not have to be applied either, if we have established that an expression subsuming other expressions is satisfied, since the subsumed expressions will be satisfied as well. In this section, we exploit these observations to more advantageously organize the XPE predicates managed by our algorithm.

For two XPEs s_1 and s_2, we say that s_1 *covers* s_2 if and only if all publications that match s_2 also match s_1. As a result, if we know s_2 is matched by a publication, then we can draw the conclusion that s_1 is also matched by this publication without further evaluating it. That is, s_1 is subsumed by s_2. We find that XPEs, where one constitutes a prefix of the other one, are in a covering relation. Suppose that for two XPEs, $s_1 : pre_1 \mapsto pre_2 \mapsto, ..., \mapsto pre_m$ and $s_2 : pre_1 \mapsto pre_2 \mapsto, ..., \mapsto pre_n (1 \leq m \leq n)$, then, s_1 is the prefix of s_2. If one publication matches s_2, the occurrence determination algorithm will satisfy it, which indicates that there exists at least one correct combination, $(o^1_1, o^1_2), (o^2_1, o^2_2), ..., (o^n_1, o^n_2)$. As a result, the occurrence determination algorithm for s_1 will return a match as result as well, since there exists at least the combination, $(o^1_1, o^1_2), (o^2_1, o^2_2), ..., (o^m_1, o^m_2)$. That is, this publication also matches s_1. Therefore, we do not need to evaluate s_1, since s_1 covers s_2. Note, the covering relation also holds, if for two expressions, one constitutes a suffix or a contained expression of the other one. We exploit prefix-covering relation in this paper and postpone others to future work.

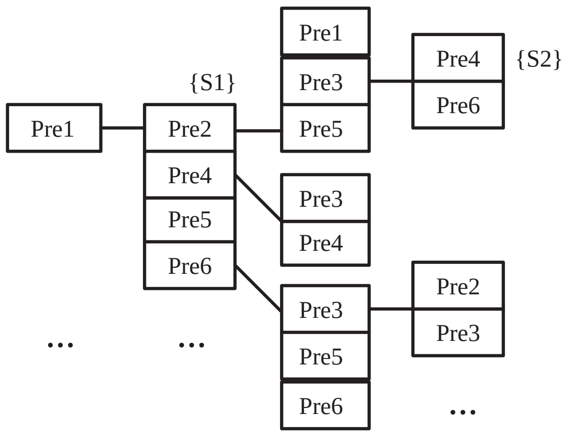


Figure 2. Fragment of XPE Index

We use an XPE data structure that takes advantage of this prefix-covering relation among expressions. A fragment of our data structure representing expressions starting with pre_1 is shown in Figure 2. Expressions are indexed by their predicates. In the figure, we designate entire XPEs through marking the final predicate with XPE identifiers (sids). For example, $s_2 : pre_1 \mapsto pre_2 \mapsto pre_3 \mapsto pre_4$ represents an XPE. s_1 is a prefix of s_2, which indicates that s_2 covers s_1. Thus, we do not need to evaluate s_1 given a publication that satisfies s_2. The strategy in processing expressions in the final stage of our algorithm is to evaluate expressions that cover the most number of expressions first. In our current implementation we approximate this through selecting the longest expression first, deferring more refined heuristics to future work.

To reduce the number of times the occurrence determination algorithm is run, we consider a further optimized XPE layout. This is based on the observation that for a set of expressions that share common predicates, if the common predicates are not satisfied, all expressions do not need to be evaluated, since all of them will be false anyway. We refer to these common predicates as *access predicates*. In our approach, we cluster all expressions by their first predicate. More refined clustering schemes are imaginable. Also, better ways of determining candidate access predicates to cluster on come to mind. However, our primary objective is to determine whether this approach can yield better results to start with. The occurrence determination only runs, if the access predicates are satisfied, otherwise, the entire cluster is ruled out. The idea of an access predicate to rule out non-matching XPath expression sub-trees in our expression index has first been proposed by Fabret *et al.* [10] in the context of matching in publish/subscribe. However, that work was not about filtering XML documents through large volumes of XPEs. Also, the access predicate was applied to subscriptions not XPE expression trees.

5 Filter Expressions

In this section we discuss how to handle attribute-based filters and nested path filters in XPEs in our approach. This illustrates the extensibility of our approach and motivates the principles for further extending our approach to other XPath language features.

Attribute-based Expressions: First, we introduce a new type of predicate, an *attribute predicate*, to handle attribute-based filters in XPEs. The predicate, $([attr, op, v])$ represents a constraint on the value of the attribute $attr$, which is attached to a tag, t, in the XPE encoding. For example, in the XPE, $s : / * /t_1[@x = 3]$, the predicate, $(p_{t_1}([x, =, 3]), =, 2)$, is used to represent not only the position information of the tag, t_1, but also the attribute and the corresponding value information of the tag t_1. The attribute predicate, $([attr, op, v])$, can be attached to any tag name variable, p_t, in the predicates (p_t, op, v), $(d(p_{t_1}, p_{t_2}), op, v)$, and $(p_t^{\dashv}, \geq, v)$.

The predicate $(p_t([attr, op_1, v_1]), op_2, v_2)$ is matched by a given tuple if and only if the tuple is $(t([attr, v_1']), v_2')$, which verifies both relations $v_1'\ op_1\ v_1$ and $v_2'\ op_2\ v_2$. For example, given a tuple, $(a([x, 6]), 5)$, which indicates that the tag a has an attribute x and corresponding value 6, we say that the predicate $(a([x, \geq, 3]), \geq, 2)$ is matched since both relations, $6 \geq 3$ and $5 \geq 2$ hold. The evaluation of attribute-based filters for the other predicates is similar.

This approach to predicate evaluation is referred to as inline evaluation of predicates, since the evaluations are performed regardless of whether the expressions are structurally matched or not. With the inline approach, the time of predicate matching increases, however, the number of matched predicates decreases, which reduces the number of times the occurrence determination algorithm has to be invoked.

An alternative approach, referred to as selection postponed, evaluates attribute predicates after the occurrence determination took place. In this case, structurally unmatched expressions are not further evaluated for attribute filter matches, only structurally matched ones are. The drawback of this approach is that after all attribute filters are matched, the occurrence determination step has to be repeated to determine whether the expression also passes the attribute filtering step. This repeated evaluation on attribute occurrences maybe traded-off by the fewer number of structurally matched expressions. We show interesting trade-offs between these two approaches in our experimental evaluation. The terms inline and selection postponed were first used by YFilter in the context of XML filtering. However, their NFA-based approach is completely different from our predicate-based technique.

Nested Path Expressions: A nested path filter is an XPE containing other XPEs in the location steps. The resulting structure of the expression is no longer a path, but a tree. These nested path expressions must be evaluated in the context of the node, which contains them.

Inspired by the *query decomposition* techniques proposed in numerous related approaches, such as XFilter and XTire [2, 5], we first decompose an XPE into several sub-expressions. For example, the XPE, $s : /a[*/c[d]/e]//c[d]/e$, can be decomposed into four sub-expressions shown in Figure 3, where each leaf node is a sub-expression. s contains two levels of decomposition since its nested paths themselves also contain nested path expressions. In the first level of decomposition, s contains two nested paths "$*/c[d]/e$" and "d" enclosed with "[]". $a//c/e$ is extracted as the main sub-expression, $/a/ * /c[d]/e$ and $/a//c/d$ are two extended sub-expressions — extending from the main expression. For each extended sub-expression, predicate, $(pos, =, v)$, is used to record the position of the last shared element with the main sub-expression. For example, $(pos, =, 2)$, indicates that $/a//c/d$ branches from the main sub-expression $/a//c/e$ at position 2. In the same way, we decompose $/a/ * /c[d]/e$ into a main sub-expression $/a/ * /c/e$ and an extended sub-expression $/a/ * /c/d$, and record their structure relation. We encode sub-expressions into ordered sets of predicates and process them separately. In order to identify whether an expression is matched after each of its sub-expressions has been evaluated, we also need to record structure information for XML documents, as shown in Figure 4. We extract XML paths, and for each path, $e = (t_1, t_2, ..., t_n)$, we record structure information with tuple, $< m_1, m_2, ..., m_n >$, where m_k represents that t_k is the m_k-th child of t_{k-1} in the XML tree.

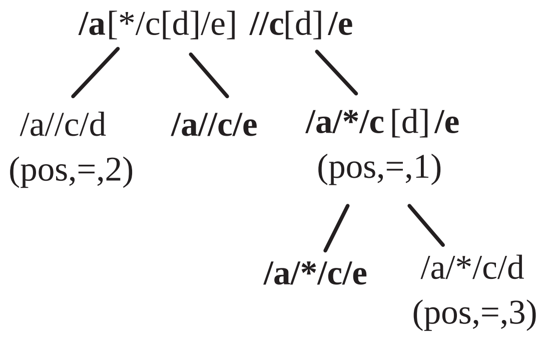


Figure 3. Decomposition of XPath expression

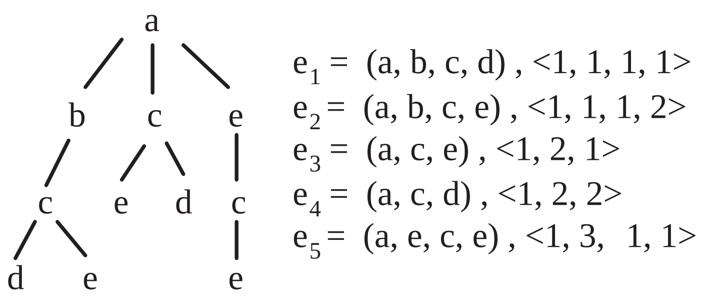

a
b c e
c e d c
d e e
e1 = (a, b, c, d) , <1, 1, 1, 1>
e2 = (a, b, c, e) , <1, 1, 1, 2>
e3 = (a, c, e) , <1, 2, 1>
e4 = (a, c, d) , <1, 2, 2>
e5 = (a, e, c, e) , <1, 3, 1, 1>

Figure 4. XML tree

Given the XPE matching results for the XPEs and the above structure information of the current XML document and XPE, we can determine whether an XPE is matched by an XML document. Consider the examples in Figure 3 and 4 again, for each sub-expression in Figure 3, we know which XML path it matches after the expression matching phase, as shown in Figure 5. For example, $/a/*/c/d$ matches $e_1 : < \underline{1}, \underline{1}, \underline{1}, \underline{1} >$, where the underlining of each number means that the matching occurs at this point. $/a//c/d$ also matches e_1, however, the matching result is $e_1 : < \underline{1}, 1, \underline{1}, \underline{1} >$, which indicates that the matching does not occur at the position 2 in the XML path. We can determine whether a true match exists or not by checking the above matching results bottom-up in the XPE decomposition tree (see Figure 5). In each level of the XPE decomposition, we compare the matching results of each extended sub-expression against the main sub-expression to determine whether a pair of XML paths exists that satisfy the position relation. For example, for $/a/*/c/e$ and $/a/*/c/d$, position relation predicate, $(pos, =, 3)$, indicates that these XPEs should have the same match (i.e., the same numbers up to and including the underlined number) before position 3 and show a difference (i.e., different numbers) after position 4, since $/a/*/c/d$ branches from $/a/*/c/e$ at position 3. For example, $< \underline{1}, \underline{1}, \underline{1}, \underline{2} >$ and $< \underline{1}, \underline{1}, \underline{1}, \underline{1} >$ satisfy the position relation; however, $< \underline{1}, \underline{3}, \underline{1}, \underline{1} >$ and $< \underline{1}, \underline{1}, \underline{1}, \underline{1} >$ do not satisfy the position relation, since they differ in the matching result starting at position 2. Thus, the matching result of non-leaf node, $/a/*/c[d]/e$, is produced and compared to nodes in its own level in the XPE decomposition tree, as indicated by the black arrows in Figure 5. $< \underline{1}, \underline{1}, \underline{1}, \underline{1 \wedge 2} >$ indicates that $/a/*/c[d]/e$ actually matches a bifurcate structure in the XML tree. Similarly, the final matching results, e_4 for $/a//c/d$, e_3 for $/a//c/e$, e_2 for $/a/*/c/e$, and e_1 for $/a/*/c/d$, can be achieved, as shown in Figure 5.

6 Evaluation

6.1 Experimental Setup

In this section, we experimentally evaluate the performance of our algorithms based on the above described encoding. We also evaluate the two ways of organizing XPath expression predicates — prefix covering and access predicate. We compare our algorithms with two popular XML filtering techniques — YFilter [8] and Index-Filter [4]. We chose YFilter, as it has been shown to have superior performance over alternative ap-

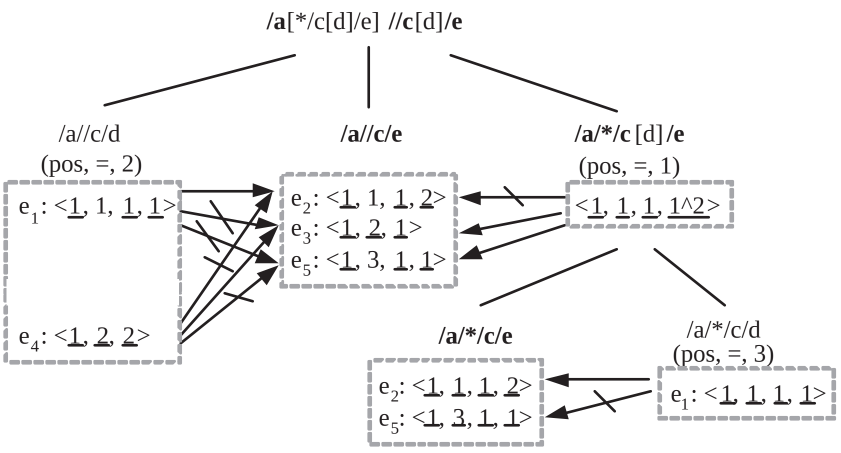


Figure 5. XPath expression matching results

proaches [8]. Yfilter is also representative for the class of automaton-based approaches. We chose Index-Filter, as one representative example of algorithms from the class of index-based filtering techniques. All algorithms are implemented in C. We modify the Index-Filter algorithm to stop after determining one match. The original algorithm is applied to find all matches in the XML document for each XPE. We perform all experiments on a computer with an Intel Xeon 2.4GHz processor with 2GB RAM running Linux Redhat 8.

For generating the XPE workload, we used the XPath generator released by Diao *et al.* [8]. For generating the XML document workload, we used the IBM XML Generator [1]. This generator was used with default parameters except that we varied the maximum number of levels of the resulting XML documents from 6 to 10. This was set consistently with the maximum length of XPEs. In order to generate these workloads, we used two different DTDs: the NITF (News Industry Text Format) DTD [15] and the PSD (Protein Sequence Database) DTD [16]. For each DTD, we generated 500 XML documents. The average size of our XML document is around 8.77 KB containing on average 140 tags. For each algorithm, we filtered all 500 XML documents against the corresponding XPath workload. All reported results are averaged over this set. All experiments report the total filtering time as the main performance metric. This time includes the time of parsing the XML document (for our algorithms this also includes the time to generate the encodings), matching the document against the XPE workload and collecting all results. As we show later, parsing time is negligible, unlike the results reported by other researchers. In our experiments, we suppose that all XPEs are processed before any XML documents are matched. That is, the time to process XPEs is not included in the filter time. XPath insertion time is an interesting metric, but not considered here. Note, in our approach, all insertion operations are constant time and the number of predicates encoding an XPE is linear in the number of location steps in an expression.

6.2 Varying the Number of XPEs

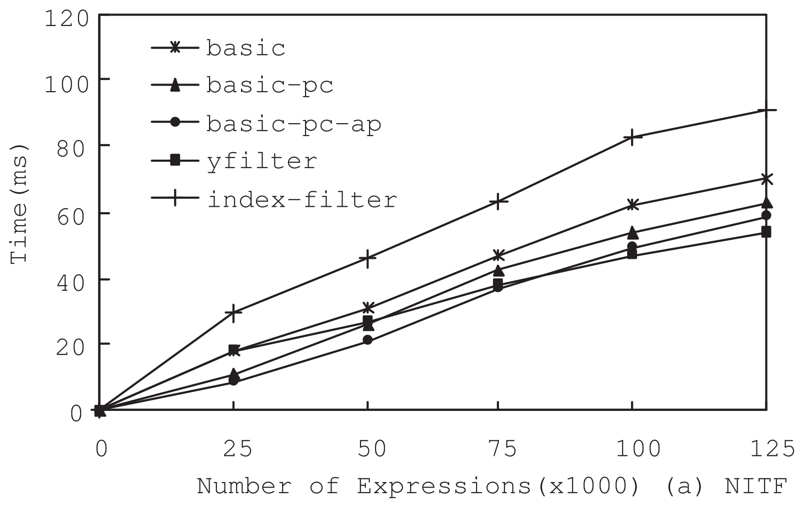
We first evaluate the scalability of the algorithms by varying the number of XPEs. First, we look at distinct expressions and then at workloads that contain duplicate expressions.

Distinct Expression Workload:

In this experiment, first, we compare three of our predicate-based filtering algorithm variants against each other to determine the best one under the given workloads. The basic matching algorithm without any optimizations (basic), the matching algorithm with prefix covering (basic-pc) and the matching algorithm with prefix covering and access predicate (basic-pc-ap). We generate two XPE workloads for NITF and PSD, respectively, and set flag of distinct vs. non-distinct (D) to $true$, maximum length of XPE (L) to 6, probability of "$*$" occurring at a location step (W) and probability of "$//$" occurring at a location step (DO) to 0.2, respectively.

The number of distinct expressions ranges from 25,000 to 125,000 and from 1,000 to 10,000 for NITF and PSD, respectively. The results for NITF are shown in Figure 6(a). As can be seen, each algorithm scales linearly with the number of distinct expressions processed. The algorithm basic performs worst. The algorithm basic-pc performs consistently better. The improvement is a direct consequence of the prefix covering technique.

The prefix covering technique is sensitive to the number of covering relations in the expressions. It is also sen-



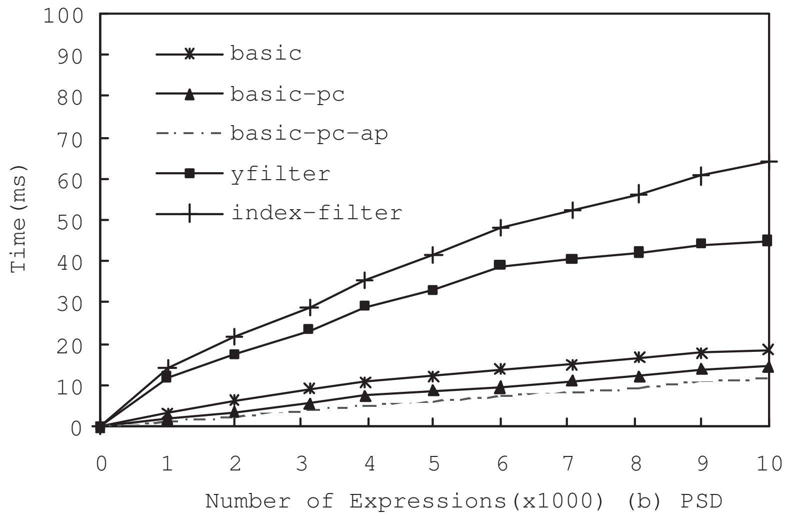


Figure 6. Varying the number of distinct XPEs (Each data point reports an average over 500 documents)

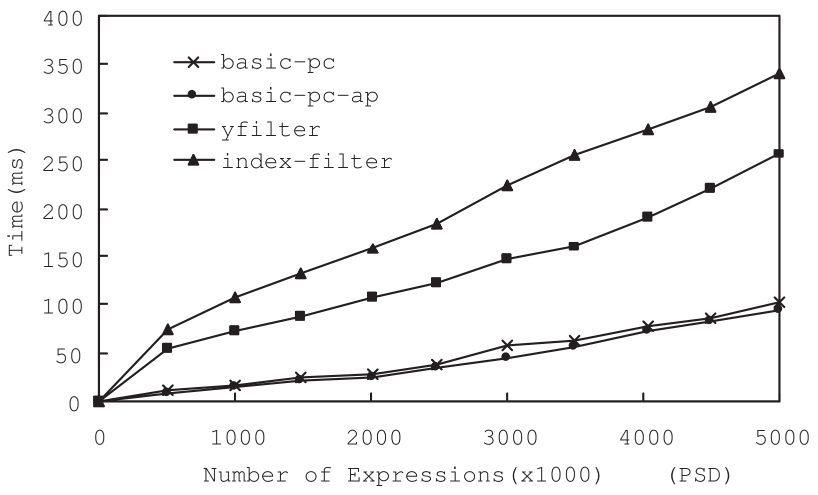


Figure 7. Workloads Including Non-distinct Expressions

sitive to the overall percentage of matched expressions. The higher the percentage of matched expressions, the fewer times the occurrence determination algorithm is expected to execute. In the NITF workload, the generated expressions are extremely selective: the percentage of matched expressions is on average 6 %. Therefore, the prefix covering technique can only contribute little, even if there is a large number of covering relations among expressions. If, in addition, we apply the access predicate technique in the basic-pc-ap version of the algorithm, we obtain the best results for this workload.

Figure 6(b) presents the results for the PSD workload. The overall trends observed above are similar for this workload. However, the main difference is the much higher percentage of match, which is 75% for this data set. Here, the prefix covering technique significantly contributes to reducing the processing time.

In Figure 6(a) and (b) we also compare our algorithms with the alternative approaches. We find that for a low percentage of matching in the workload (6 % in NITF in Figure 6(a)) YFilter performs better than our basic-pc-ap algorithm, if the number of expressions is greater than 100,000. YFilter can take better advantage of the high selectivity of the NITF workload. If only a small number of expressions are matched, the execution of the NFA extends to only a very limited number of states as the XML tag elements do not trigger new transitions. In contrast, our algorithm translates all the XML paths into attribute-value pairs and matches them against the existing predicates, whether the paths match any predicates or not. The Index-Filter algorithm performs worst and takes almost twice as much time as YFilter. Our workload is not favorable for Index-Filter. Its strength is the processing of small number of expressions over large XML documents.

However, in a contrasting scenario, where the percentage of matching in the workload is high, as for the PSD DTD in Figure 6(b) with 75 % matching, the difference between YFilter and our algorithms becomes much more significant. As compared to the above case, the situation is reversed with our algorithms performing significantly better. The YFilter NFA has to undergo a large number of transitions and touches a lot of states, whereas our algorithm amortizes the work done in predicate matching and XPE matching in determining the large number of matches. Index-Filter still performs least favorable, as explained above.

Duplicate Expression Workload: In large, Internet-scale filtering system the expression workloads are likely to contain duplicate expressions. These duplicates represent the common and shared interests among users. In this experiment, we vary the number of expressions from half a million to 5 millions and set D to $false$, all other parameters are the same as before. The number of distinct expressions ranges from 55,000 to 167,000 and from 5,500 to 10,000 for NITF and PSD DTDs, respectively. Figures 7 shows the results for PSD DTD. The NITF results are similar with distinct expression experiment. For both DTDs, all algorithms scale linearly when increasing the number of expressions. In the NITF workload, our algorithm performs slightly better than YFilter. basic-pc-ap and yfilter perform 121.5 and 137.3 ms over 5 millions of duplicate expressions, respectively, which indicates that our algorithm accommodates duplicate expressions in the workload slightly better. For the PSD DTD, our algorithm outperforms YFilter in the larger expression regimes by more than half of its matching time. Again, this is due to the high percentage of matching.

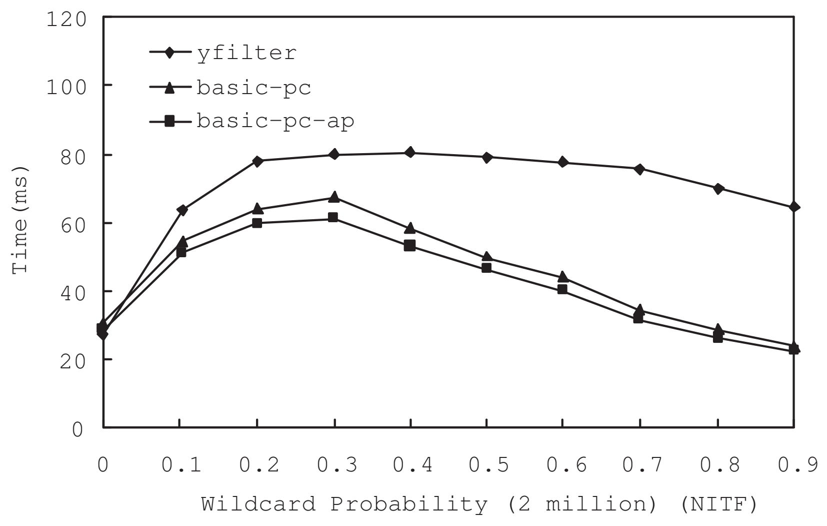


Figure 8. Varying the wildcard probability

6.3 Effect of Wildcards and Descendants

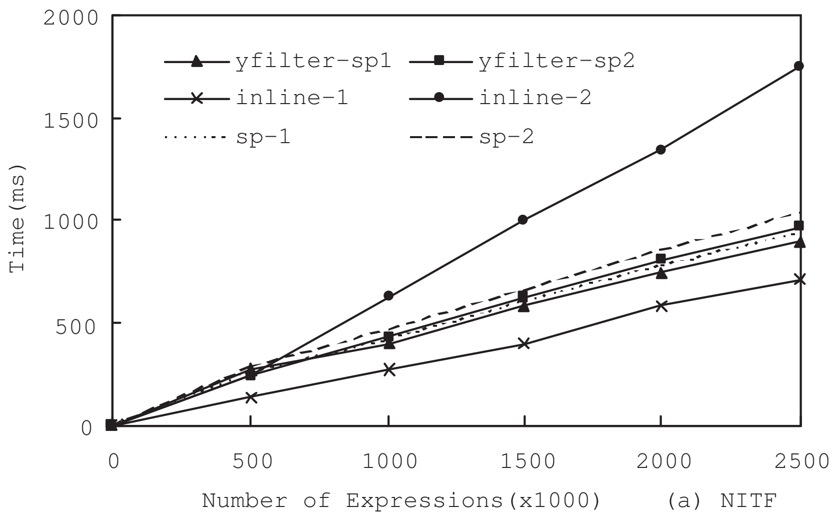
In this section, we investigate the impact of the probability of wildcards and descendant operators on matching time. We only report the NITF workload, since the PSD workload has the similar results. For wildcards, we set DO to 0.2 and vary W from 0 to 0.9; for descendant operators, we set W to 0.2 and vary DO from 0 to 0.9. Both workloads contain 2 million expressions. We show the NITF results for increasing wildcard in Figure 8. The results for varying descendant operator probability are similar. With the increase in wildcard probability, the number of predicates first grows because the addition of wildcards will increase the number of predicates, adding new predicates with increasing range values to the existing predicates. As W is further increased, the overall number of predicates begins to decrease resulting in fewer distinct predicates. The turning point here is 0.3. After this point, the expressions become very similar to each other and more overlapping and duplicate expressions are in the workload resulting in less matching time. For descendent operator, with increasing probability, the number of predicates first increases because of the addition of further relative, *greater-than-or-equal* predicates and, finally, a decrease because expressions overlap and more duplicates come about in the workload.

In YFilter, the increase of W and DO will increase the non-determinism of the underlying NFA. Even when W is set to 0.9, at which point the number of distinct expressions has significantly decreased, YFilter's performance does not improve significantly like our algorithm. This happens because more states of the NFA are touched in filtering. The wildcards are matched by any XML element leading to many state transitions. A similar argument applies to varying DO.

Index-Filter is not evaluated for increasing W. The original paper does not discuss how to handle wildcards. In our implementation, we just simply let wildcards match any XML elements, which makes the size of *index stream* of each node in the prefix tree augment rapidly, especially when the probability of wildcards is high. For descendant operator, it is also less sensitive to the varying probability like YFilter.

6.4 Effect of Attribute-based Filters

In this section, we study the performance of our algorithm when the XPEs have attribute-based filters. We implement the inline and selection postponed approach based on the basic-pc-ap algorithm. For YFilter, we implement the selection postponed approach, since it has been praised as superior to an inline approach for YFilter [8]. We generate expression workloads with one and two filters per path, respectively. The NITF results are shown in Figure 9(a). In the NITF workload, most expressions are not structurally matched. Therefore, selection postponed approaches, in both our algorithm and YFilter, are less sensitive to the increase in the number of filters per expression. The filters are checked only if the expressions are structurally matched. Our inline approach responds to two factors. One, the number of attributes in the XML document and, two, the



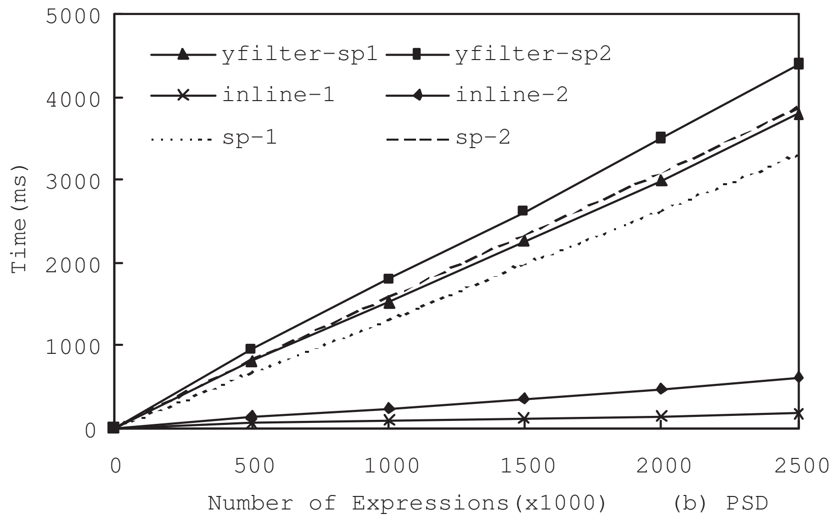


Figure 9. Varying the number of filters per expression

number of filters in the expression. The NITF XML documents contain more attributes than the PSD ones. As a result, the difference between inline-1 (one attribute per path) and inline-2 (two attributes per path) becomes insignificant in the PSD results, since the algorithm needs less time to evaluate these attributes in the document. As expected, our algorithms based on the inline approach perform better than YFilter on the PSD workload because of the high percentage of matching. YFilter has to evaluate more filters after the structural matching in the NITF workload. However, the selection postponed technique in the context of our approach does not yield good performance for the PSD workload since the high percentage of matching results leads to more applications of the occurrence determination algorithm, when we evaluate the attribute predicates.

6.5 Matching Cost Breakdown

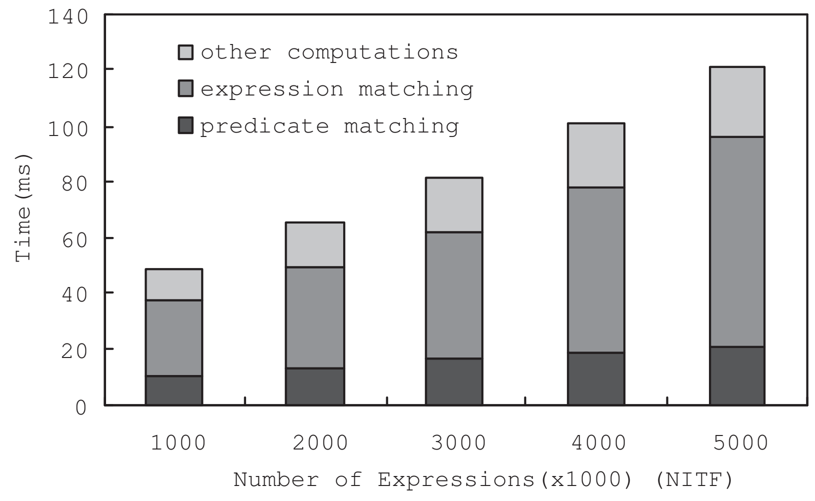


Figure 10. Cost of predicate and expression matching

In this experiment we aim to understand where the time is spent in our algorithm. The total filtering time in all our experiments includes the time spent on parsing the XML document, performing predicate matching and performing expression matching. In this experiment we isolate these times to understand the impact of the different stages of the algorithm. Our evaluations show that parsing time is negligible compared to the total filtering time. The average parsing time for NITF and PSD XML documents is only 314 and 355 microseconds, respectively. We, therefore, do not plot these numbers, as they are too small. In the experiment, we used the same workload as in the experiment with duplicate expressions. Figure 10 shows the cost breakdown for predicate matching, expression matching (i.e., occurrence determination), and other computations (i.e., collecting the results). We only plot NITF workload results (PSD results are similar). As can be seen in the figure, the cost for expression matching dominates. As the number of expression increases, the time spent on predicate matching rises less severely than the time spent on expression matching. As more expressions are processed, the number of distinct predicates does not increase to the same degree as the number of expressions. This is because our algorithm takes advantage of duplicated predicates storing and processing them only once. The numbers of distinct predicates that are managed in our algorithm for the plotted data points are 4019, 4593, 5150, 5521, and 5843, which changes comparatively little compared to the change in expression numbers (i.e., from 1000 K to 5000 K).

7 Conclusions

In this paper, we have developed and evaluated a novel matching algorithm to solve the XML/XPath filtering problem for filtering XML documents against large numbers of XPEs. The novel ideas underlying our algorithm are to encode XPEs as Boolean predicates, encode XML documents as sets of (attribute, value)–pairs, and efficiently evaluate these tuples over the Boolean predicates. This constitutes an altogether new approach for solving the XML filtering problem, next to the popular automaton-based and index-based filtering algorithms.

The strength of our approach is the great potential to take advantage of overlap in different XPEs. The overlap exploitation is manifest in the predicate encoding that stores and processes distinct predicates once. We have experimentally evaluated the performance of our algorithm and have constructively proven its scalability for processing of Internet-scale workloads (millions of XPEs and matching performance in the millisecond range). We have evaluated our approach next to common alternatives, namely YFilter and Index-Filter. This comparison gave rise to interesting trade-offs and novel insights.

For an XPE workload with a high percentage of matched expressions, our algorithm is more efficient than the alternative approaches investigated. For a contrasting workload with a very low matching percentage, YFilter outperformed our algorithm. For workloads that are based on expressions that are characterized by many wildcards and descendant operators, our algorithm consistently outperformed alternatives. For expression workloads including attribute filters and high matching percentages, our inline predicate evaluation algorithm outperformed alternatives. In contrast, the selection postponed approaches performed better for an expression workload with a low matching percentage. All selection postponed approaches performed within a small variance of each other.

The proposed algorithm is thus a completely novel and viable alternative to existing automaton-based and index-based approaches.

References

[1] D. A.L.Diaz. XML generator, Sept. 2003.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th VLDB Conference*, 2000.

[3] J. Bitner and E. M. Reingold. Backtracking programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[4] N. Bruno, L. Gravano, N. Doudas, and D. Srivastava. Navigation-vs. index-based XML multi-query processing. In *Proceeding of ICDE*, 2003.

[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. Chen, S. Davidson, and Y. Zheng. Blas: An efficient xpath processing system. In *Proceedings of the 2004 ACM SIGMOD international conference on on Management of data*, pages 47–58. ACM Press, 2004.

[7] R. Dechter and D. Frost. Backtracking algorithms for constraint satisfaction problems. *Technical Report*, 1999.

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

[9] Y. Diao, P. Fischer, M. Franklin, and R. To. Yfilter: Efficient and scalable filtering of XML documents. In *Proceedings of ICDE2002*, 2002.

[10] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. Ross, and D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe systems. In *SIGMOD Conference*, 2001.

[11] D. Florescu and D. Kossmann. Storing and querying XML data using an RDBMS. In *Bulletin of the Technical Committee on Data Engineering*, pages 27–34, September 1999.

[12] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*, pages 173–189. Springer-Verlag, 2002.

[13] A. K. Gupta and D. Suciu. Stream processing of xpath queries with predicates. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 419–430. ACM Press, 2003.

[14] L. V. S. Lakshmanan and S. Parthasarathy. On efficient matching of streaming XML documents and queries. In *Extending Database Technology*, pages 142–160, 2002.

[15] http://www.nitf.org.

[16] http://pir.georgetown.edu.

[17] J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh, and B. Reinwald. Efficiently Publishing Relational Data as XML Documents. *VLDB J.*, 10(2-3):133–154, 2001.

[18] F. Tian, B. Reinwald, H. Pirahesh, T. Mayr, and J. Myllymaki. Implementing a scalable XML publish/subscribe system using relational database systems. In *SIGMOD*, 2004.

[19] http://www.w3.org/2002/ws.

[20] http://www.w3.org/tr/xpath20.

A Proof of the Correctness of Predicate Encoding

In the predicate-based filtering of XPath expressions against XML document, we encode XPath expression as ordered set of predicates and translate XML document into set of tuples. In this appendix, we prove the correctness of our encoding. XPath expression matches XML document path if and only if the corresponding XPath expression encoding (ordered set of predicates) matches the corresponding XML document path encoding (set of tuples).

A.1 Introduction

Each XPath expression is translated into an *ordered set of predicates*, where each predicate is an (attribute, operator, value)–triple that expresses a constraint over the value of the attribute (i.e., the position of the tag). The predicate can be one of the following four types: absolute predicate, relative predicate, end-of-path predicate and length-of-expression predicate.

In our approach, we use the common interpretation of an XML document as a tree of nodes and consider each path from the root node in this tree to a leaf node, separately. Thus, we decompose each XML document into a set of XML paths and each path is encoded as a set of (attribute-value)-pairs. For example, the XML path, $e = (p_1, p_2, ..., p_n)$, in our encoding is $\{(length, n), (p_1, 1), (p_2, 2), ..., (p_n, n)\}$.

We will prove the correctness of our encoding in the following section by induction.

A.2 Proof

Theorem A.1 *Let $s = op_1\ t_1\ op_2\ t_2\ ......\ op_n\ t_n$ be an XPath expression, where op_i $(1 \leq i \leq n)$ is either parent-child or ancestor-descendant operator, t_i $(1 \leq i \leq n)$ is either a tag name or wildcard and n is the length of XPath expression. Let $e = (p_1, p_2, ..., p_m)$ be an XML path consisting of a sequence of tag names, p_i, and m is the length of XML document path. Let s' be the predicate-based encoding of s and e' be the document path encoding of e. We say that s matches e iff s' matches e'.*

Proof:

"IF" : We will prove that if s matches e then s' matches e' under our encoding. We show by induction on the length of s, n. First, we prove that Theorem A.1 holds when $n = 1$. Second, assume it holds for $n = k$, then, we prove that it holds when $n = k + 1$.

Let us consider $n = 1$, s can be $/t$, $//t$, $/*$ and $//*$. In all four cases, we prove that Theorem A.1 is true by following the steps listed below:

1. Construct general format of e, which matches the specified s.
2. Translate s and e into s' and e', respectively, with our encoding.
3. Prove that s' matches e' under our evaluation rules.

Case 1 : $s = /t$. First, $e = (t, p_1, p_2, ..., p_q)$. Second, $s' = (p_t, =, 1)$ and $e' = \{(length, q+1), (t, 1), (p_1, 2), ..., (p_q, q+1)\}$. Third, $(p_t, =, 1)$ matches $(t, 1)$ in e'.

Case 2 : $s = t$. First, $(p_1, ..., p_{i-1}, t, p_i, ..., p_q)$ $(1 \leq i \leq q)$. Second, $s' = (p_t, \geq, 1)$ and $e' = \{(length, q+1), (p_1, 1), ..., (p_{i-1}, i-1), (t, i), (p_i, i+1), ..., (p_q, q+1)\}$. Third, $(p_t, \geq, 1)$ matches (t, i) in e' since $i \geq 1$.

Case 3 : $s = /*$. First, $e = (p_1, p_2, ..., p_q)$ $(q \geq 1)$. Second, $s' = (length, \geq, 1)$ and $e' = \{(length, q), (p_1, 1), ..., (p_q, q)\}$. Third, $(length, \geq, 1)$ matches $(length, q)$ since $q \geq 1$.

Case 4 : $s = *$. It is same with Case 3, since we do not distinguish them in our matching semantic.

Let us assume that Theorem A.1 holds for $n = k$, that is, if $s_k = op_1\ t_1\ op_2\ t_2\ ......\ op_k\ t_k$ matches $e_k = (p_1, p_2, ..., p_q)$ (the subscript k is not for the length of e, but represents a matched path for s_k), then $s'_k = pre_1 \mapsto pre_2 \mapsto ... \mapsto pre_l$ matches $e'_k = \{(length, q), (p_1, 1), ..., (p_q, q)\}$. We will prove it holds for $n = k + 1$. First of all, we assume that s_k actually matches part of e_k, $(p_i, p_{i+1}, ..., p_j)$ $(1 \leq i < j \leq q)$. Thus, t_1 matches p_i and t_k matches p_j, respectively, and $j - i + 1 \geq k$ holds since any op in s_k can be $//$-operator. Given the assumption, there are several cases need to be considered. For each of them, we prove that Theorem A.1 holds for $n = k + 1$ by following the steps listed below:

1. Construct s_{k+1} based on s_k.
2. Construct general format of e_{k+1}, which matches s_{k+1}.
3. Translate s_{k+1} and e_{k+1} into s'_{k+1} and e'_{k+1}, respectively, with our encoding.

4. Prove that s'_{k+1} matches e'_{k+1} under our evaluation rules.

Case 1 : t_k is a tag name and thus $t_k = p_j$. $op_{k+1}\ t_{k+1}$ of s_{k+1} can be $/t$, $//t$, $/*$ and $//*$. We discuss them respectively.

- $op_{k+1}\ t_{k+1} = /t$. First, s_{k+1} is s_k followed by $/t$, which is encoded as $s_k \cdot /t$ in our proof. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_j, t, p_{j+1}, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_l \mapsto (d(p_{t_k}, p_t), =, 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_j, j), (t, j+1), (p_{j+1}, j+2), ..., (p_q, q+1)\}$. Fourth, the new predicate $(d(p_{t_k}, p_t), =, 1)$ matches tuples (p_j, j) and $(t, j+1)$ since $t_k = p_j$ actually.

- $op_{k+1}\ t_{k+1} = //t$. First, $s_{k+1} = s_k \cdot //t$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_j, p_{j+1}, ..., t, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_l \mapsto (d(p_{t_k}, p_t), \geq, 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_j, j), (p_{j+1}, j+1), ..., (t, r), ...(p_q, q+1)\}$ $(j+1 \leq r \leq q+1)$. Fourth, the new predicate $(d(p_{t_k}, p_t), \geq, 1)$ matches tuples (p_j, j) and (t, r) since $r - j \geq 1$.

- $op_{k+1}\ t_{k+1} = /*$ or $//*$. First, $s_{k+1} = s_k \cdot /*$, or $s_k \cdot //*$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_j, t, p_{j+1}, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_l \mapsto (p^{\dashv}_{t_k}, \geq, 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_j, j), (t, j+1), (p_{j+1}, j+2), ..., (p_q, q+1)\}$. Fourth, the new predicate $(p^{\dashv}_{t_k}, \geq, 1)$ matches tuples $(length, q+1)$ and (p_j, j) since $q + 1 - j \geq 1$.

Case 2 : t_k is a wildcard and at least one tag name exists in s_k. Thus, pre_l must be an end-of-path predicate and should be removed since new end $op_{k+1}\ t_{k+1}$ will be added. Assume that t_u is the last tag name in s_k and thus all t_v $(u < v \leq k)$ are wildcards. Moreover, assume that t_u matches p_w in e_k, and thus $i \leq w < j$ and $t_u = p_w$ hold. $op_{k+1}\ t_{k+1}$ of s_{k+1} can be $/t$, $//t$, $/*$ and $//*$. We discuss them respectively.

- $op_{k+1}\ t_{k+1} = /t$ and all op_v $(u < v \leq k)$ are /-operator. First, $s_{k+1} = s_k \cdot /t$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_w, ..., p_j, t, p_{j+1}, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_{l-1} \mapsto (d(p_{t_u}, p_t), =, k - u + 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_w, w), ..., (p_j, j), (t, j+1), (p_{j+1}, j+2), ..., (p_q, q+1)\}$. Fourth, the new predicate $(d(p_{t_u}, p_t), =, k - u + 1)$ matches tuples (p_w, w) and $(t, j+1)$ since no //-operator appears between t_u and t_k, and thus $k - u + 1 = j + 1 - w$ holds.

- $op_{k+1}\ t_{k+1} = /t$ and some op_v $(u < v \leq k)$ is //-operator, or $op_{k+1}\ t_{k+1} = //t$. First, $s_{k+1} = s_k \cdot /t$, or $s_k \cdot //t$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_w, ..., p_j, t, p_{j+1}, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_{l-1} \mapsto (d(p_{t_u}, p_t), \geq, k - u + 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_w, w), ..., (p_j, j), (t, j+1), (p_{j+1}, j+2), ..., (p_q, q+1)\}$. Fourth, the new predicate $(d(p_{t_u}, p_t), \geq, k - u + 1)$ matches tuples (p_w, w) and $(t, j+1)$ since some //-operator appears between t_u and t_{k+1}, and thus $j + 1 - w \geq k - u + 1$ holds.

- $op_{k+1}\ t_{k+1} = /*$ or $//*$. First, $s_{k+1} = s_k \cdot /*$, or $s_k \cdot //*$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_w, ..., p_j, t, p_{j+1}, ..., p_q)$. Third, $s'_{k+1} = pre_1 \mapsto ... \mapsto pre_{l-1} \mapsto (p^{\dashv}_{t_u}, \geq, k - u + 1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_w, w), ..., (p_j, j), (t, j+1), (p_{j+1}, j+2), ..., (p_q, q+1)\}$. Fourth, the new predicate $(p^{\dashv}_{t_u}, \geq, k - u + 1)$ matches tuples $(length, q+1)$ and (p_w, w), since $q + 1 - w \geq k - u + 1$.

Case 3 : all t_u $(1 \leq u \leq k)$ are wildcards. Then s'_k has only one predicate, $(length, \geq, k)$. It should be removed since new end $op_{k+1}\ t_{k+1}$ will be added. $op_{k+1}\ t_{k+1}$ can be $/t$, $//t$, $/*$ and $//*$. We will discuss them respectively.

- $op_{k+1}\ t_{k+1} = /t$ and all op_u $(1 \leq u \leq k)$ are /-operator. In this case, s_k actually matches $(p_1, ..., p_k)$ in e_k. First, $s_{k+1} = s_k \cdot /t$. Second, $e_{k+1} = (p_1, ..., p_k, t, ..., p_q)$. Third, $s'_{k+1} = (p_t, =, k+1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_k, k), (t, k+1), ..., (p_q, q+1)\}$. Fourth, the predicate $(p_t, =, k+1)$ matches tuple $(t, k+1)$ obviously.

- $op_{k+1}\ t_{k+1} = /t$ and some op_u $(1 \leq u \leq k)$ is //-operator, or $op_{k+1}\ t_{k+1} = //t$. First, $s_{k+1} = s_k \cdot /t$, or $s_k \cdot //t$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_j, t, ..., p_q)$ $(1 \leq i < j \leq q)$. Third, $s'_{k+1} = (p_t, \geq, k+1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_j, j), (t, j+1), ..., (p_q, q+1)\}$. Fourth, the predicate $(p_t, \geq, k+1)$ matches tuple $(t, j+1)$, since $j + 1 \geq k + 1$.

- $op_{k+1}\ t_{k+1} = /*$ or $//*$. First, $s_{k+1} = s_k \cdot /*$, or $s_k \cdot //*$. Second, $e_{k+1} = (p_1, ..., p_i, ..., p_j, t, ..., p_q)$. Third, $s'_{k+1} = (length, \geq, k+1)$ and $e'_{k+1} = \{(length, q+1), (p_1, 1), ..., (p_i, i), ..., (p_j, j), (t, j+1), ..., (p_q, q+1)\}$. Fourth, the predicate $(length, \geq, k+1)$ matches tuple $(length, q+1)$, since $q + 1 \geq k + 1$.

"ONLY IF" : We will prove that if s' matches e' then s matches e. We show by induction on the number of predicates in s', n. First, we prove that Theorem A.1 holds for $n = 1$. Second, assume it holds for $n = k$, then we prove that it holds when $n = k + 1$.

Let us consider $n = 1$, s' can be the absolute predicate, the relative predicate, and the length-of-expression predicate. In each case, we prove that Theorem A.1 is true by following the steps listed below:

1. List tuples that e' should contain, in order to match the specified s'.
2. Construct general format of s and e based on s' and e', respectively, according to our encoding.
3. Prove that s matches e.

Case 1 : $s' = (p_t, =, v)$. First, e' should contain (t, v). Second, $s = /*/.../*/t$ (no //-operator before t and the number of $*$ is $v - 1$) and $e = (p_1, ..., p_{v-1}, t, p_v, ..., p_q)$ $(1 \leq v \leq q)$. Third, s matches e obviously.

Case 2 : $s' = (p_t, \geq, v)$. First, e' should contain (t, v') $(v' \geq v)$. Second, $s = /*/.../*/t$, or $*/.../*/t$ (//-operator can be anywhere before t and the number of $*$ is $v-1$) and $e = (p_1, ..., p_v, t, p_{v+1}, ..., p_q)$ $(1 \leq v \leq q)$. Third, s matches e since $v + 1 \geq v$.

Case 3 : $s' = (d(p_{t_1}, p_{t_2}), =, v)$. First, e' should contain tuples (t_1, v_1) and (t_2, v_2) $(v_2 - v_1 = v)$. Second, $s = t_1/*/.../*/t_2$ (no //-operator between t_1 and t_2 and the number of $*$ is $v - 1$) and $e = (p_1, ..., p_{v_1-1}, t_1, p_{v_1}, ..., p_{v_2-2}, t_2, p_{v_2-1}, ..., p_q)$ $(1 \leq v_1 < v_2 \leq q)$. Third, s matches e since $(v_2 - 1) + 1 - v_1 = v$.

Case 4 : $s' = (d(p_{t_1}, p_{t_2}), \geq, v)$. First, e' should contain tuples (t_1, v_1) and (t_2, v_2) $(v_2 - v_1 \geq v)$. Second, $s = t_1/*/.../*/t_2$ (//-operator can be anywhere between t_1 and t_2 and the number of $*$ is $v - 1$) and $e = (p_1, ..., p_{v_1-1}, t_1, p_{v_1}, ..., p_{v_2-1}, t_2, p_{v_2}, ..., p_q)$ $(1 \leq v_1 < v_2 \leq q)$. Third, s matches e since $v_2 + 1 - v_1 \geq v$.

Case 5 : $s' = (length, \geq, v)$. First, e' should contain tuple $(length, v')$ $(v' \geq v)$. Second, $s = */.../*$ or $/*/.../*$ (//-operator can be anywhere in s and the number of $*$ is v) and $e = (p_1, ..., p_{v+1})$. Third, s matches e since $v + 1 \geq v$.

Let us assume that Theorem A.1 holds for $n = k$, that is, if $s'_k = pre_1 \mapsto pre_2 \mapsto ... \mapsto pre_k$ matches $e'_k = \{(length, q), (p_1, 1), ..., (p_q, q)\}$, then $s_k = op_1\ t_1\ op_2\ t_2\ ......\ op_l\ t_l$ matches $e_k = (p_1, p_2, ..., p_q)$. We will prove it holds for $n = k+1$. Given the assumption of $n = k$, there are several cases need to be considered. For each of them, we prove that Theorem A.1 holds for $n = k + 1$ by following the steps listed below:

1. Construct s'_{k+1} based on s'_k.
2. List tuples that e'_{k+1} should contain, in order to match s'_{k+1}.
3. Construct general format of s_{k+1} and e_{k+1} based on s'_{k+1} and e'_{k+1}, respectively, according to our encoding.
4. Prove that s_{k+1} matches e_{k+1}.

Case 1 : pre_k is an absolute predicate. In this case, k can only be 1 under our predicate language since absolute predicate must be the first predicate in the encoding. Thus, $s'_k = (p_{t_1}, op_1, v_1)$, where op_1 can be either $=$ or $\geq$. The e'_k should contain (t_1, v') $(v'\ op_1\ v_1)$ in order to match s'_k. The corresponding e_k should be $(p_1, ..., p_{v'-1}, t_1, p_{v'}, ..., p_q)$. The pre_{k+1} can be either relative or end-of-path predicate. We discuss them respectively.

- pre_{k+1} is a relative predicate. For $pre_{k+1} = (d(p_{t_1}, p_{t_2}), =, v)$, first, $s'_{k+1} = (p_{t_1}, op_1,\ v_1) \mapsto (d(p_{t_1}, p_{t_2}), =, v)$. Second, e'_{k+1} should contain tuples (t_1, v') and (t_2, v_2) $(v_2 - v' = v)$. Third, $s_{k+1} = s_k \cdot /*/.../*/t_2$ (no //-operator between s_k and t_2 and the number of $*$ between s_k and t_2 is $v - 1$), and $e_{k+1} = (p_1, ..., p_{v'-1}, t_1, p_{v'}, ..., p_{v_2-2}, t_2,\ p_{v_2-1}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $(v_2-1)+1-v' = v$. For $pre_{k+1} = (d(p_{t_1}, p_{t_2}), \geq, v)$, first, $s'_{k+1} = (p_{t_1}, op_1, v_1) \mapsto (d(p_{t_1}, p_{t_2}), \geq, v)$. Second, e'_{k+1} should contain tuples (t_1, v') and (t_2, v_2) $(v_2 - v' \geq v)$. Third, $s_{k+1} = s_k \cdot /*/.../*/t_2$ (//-operator can be anywhere between s_k and t_2 and the number of $*$ between s_k and t_2 is $v - 1$), and $e_{k+1} = (p_1, ..., p_{v'-1}, t_1, p_{v'}, ..., p_{v_2-1}, t_2, p_{v_2}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $v_2+1-v' \geq v$.
- pre_{k+1} is an end-of-path predicate, that is, $pre_{k+1} = (p^{\dashv}_{t_1}, \geq, v)$. First, $s'_{k+1} = (p_{t_1}, op_1, v_1) \mapsto (p^{\dashv}_{t_1}, \geq, v)$. Second, e'_{k+1} should contain tuples (t_1, v') and $(length, q + 1)$ $(q + 1 - v' \geq v)$. Third,

$s_{k+1} = s_k \cdot / * /.../*$ (//-operator can be anywhere after s_k and the number of $*$ after s_k is v), and $e_{k+1} = (p_1, ..., p_{v'-1}, t_1, p_{v'}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $q + 1 - v' \geq v$.

Case 2 : pre_k is a relative predicate, that is, $pre_k = (d(p_t, p_{t_1}), op_1, v_1)$, where op_1 can be either $=$ or $\geq$. The e'_k should contain tuples (t, x) and (t_1, y) $(y - x\ op_1\ v_1)$, and corresponding e_k should be $(p_1, ..., p_{x-1}, t, p_x, ..., p_{y-2}, t_1, p_{y-1}, ..., p_q)$ $(1 \leq x < y \leq q)$. The pre_{k+1} can be either relative or end-of-path predicate. We discuss them respectively.

- pre_{k+1} is a relative predicate. For $pre_{k+1} = (d(p_{t_1}, p_{t_2}), =, v)$, first, $s'_{k+1} = s'_k \mapsto (d(p_{t_1}, p_{t_2}), =, v)$. Second, e'_{k+1} should contain tuples (t_1, y) and (t_2, v_2) $(v_2 - y = v)$. Third, $s_{k+1} = s_k \cdot / * /.../ * /t_2$ (no //-operator between s_k and t_2 and the number of $*$ between s_k and t_2 is $v - 1$), and $e_{k+1} = (p_1, ..., p_{x-1}, t, p_x, ..., p_{y-2}, t_1, p_{y-1}, ..., p_{v_2-3}, t_2, p_{v_2-2}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $(v_2 - 2) + 2 - y = v$. For $pre_{k+1} = (d(p_{t_1}, p_{t_2}), \geq, v)$, first, $s'_{k+1} = s'_k \mapsto (d(p_{t_1}, p_{t_2}), \geq, v)$. Second, e'_{k+1} should contain tuples (t_1, y) and (t_2, v_2) $(v_2 - y \geq v)$. Third, $s_{k+1} = s_k \cdot / * /.../ * /t_2$ (//-operator can be anywhere between s_k and t_2 and the number of $*$ between s_k and t_2 is $v - 1$), and $e_{k+1} = (p_1, ..., p_{x-1}, t, p_x, ..., p_{y-2}, t_1, p_{y-1}, ..., p_{v_2-2}, t_2, p_{v_2-1}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $v_2 + 1 - y \geq v$.

- pre_{k+1} is an end-of-path predicate, that is, $pre_{k+1} = (p^{\dashv}_{t_1}, \geq, v)$. First, $s'_{k+1} = s'_k \mapsto (p^{\dashv}_{t_1}, \geq, v)$. Second, e'_{k+1} should contain tuples (t_1, y) and $(length, q + 1)$ $(q + 1 - y \geq v)$. Third, $s_{k+1} = s_k \cdot / * /.../*$ (//-operator can be anywhere after s_k and the number of $*$ after s_k is v), and $e_{k+1} = (p_1, ..., p_{x-1}, t, p_x, ..., p_{y-2}, t_1, p_{y-1}, ..., p_q)$. Fourth, s_{k+1} matches e_{k+1} since $q + 1 - y \geq v$.

Note, pre_k can neither be a length-of-expression predicate nor an end-of-path predicate since the length-of-expression indicates that the s'_k should only contain one predicate and no more predicate can be added, and the end-of-path indicates that the end of path has been touched.