AFilter: Adaptable XML Filtering with Prefix-Caching and Suffix-Clustering

K. Selçuk Candan Wang-Pin Hsiung Songting Chen Junichi Tatemura Divyakant Agrawal
NEC Laboratories America
10080 North Wolfe Road, Suite SW3-350
Cupertino, CA 95014
{candan,whsiung,songting,tatemura,agrawal}@sv.nec-labs.com

ABSTRACT

XML message filtering problem involves searching for instances of a given, potentially large, set of patterns in a continuous stream of XML messages. Since the messages arrive continuously, it is essential that the filtering rate matches the data arrival rate. Therefore, the given set of filter patterns needs to be indexed appropriately to enable real-time processing of the streaming XML data. In this paper, we propose *AFilter*, an *adaptable*, and thus scalable, path expression filtering approach. AFilter has a *base* memory requirement linear in filter expression and data size. Furthermore, when additional memory is available, AFilter can exploit prefix commonalities in the set of filter expressions using a loosely-coupled prefix caching mechanism as opposed to tightly-coupled active state representation of alternative approaches. Unlike existing systems, AFilter can also exploit suffix-commonalities across filter expressions, while simultaneously leveraging the prefix-commonalities through the cache. Finally, AFilter uses a triggering mechanism to prevent excessive consumption of resources by delaying processing until a *trigger* condition is observed. Experiment results show that AFilter provides significantly better scalability and runtime performance when compared to state of the art filtering systems.

1. INTRODUCTION

XML message filtering systems are used for sifting through real-time messages to support publish/subscribe [11, 25, 26], real-time business data mining, accounting, and reporting for enterprises. A filtering system continuously evaluates a given set of registered filter predicates on real-time message streams to identify the relevant data for higher-level processing. Thus, XML filtering problem is concerned with finding instances of a given, potentially large, set of patterns in a continuous stream of data trees (or XML messages). Specifically, if $\{x_1, x_2, ...\}$ denotes a stream of XML messages, where x_i is i^{th} XML message in the stream, and $\{q_1, \ldots, q_m\}$ is a set of filter predicates (described in an XML query language, such as XPath or XQuery [1]) then an XML filtering system identifies (in real-time) $\langle x_i, q_j, PT_{ij} \rangle$ triplets, such that the message x_i satisfies the filter query q_j. The set PT_{ij} includes each instantiation of the query (referred to as path-tuples in [14]).

VLDB '06, September 12-15, 2006, Seoul, Korea.

1.1 Existing Approaches and Challenges

Obviously, the XML filtering problem is related to, but different from, the more traditional stored XML data *retrieval problem*, where given a stored collection of XML data objects and a query, the system needs to identify those data instances which satisfy the given query. XML query processing approaches concentrate on finding effective mechanisms for matching query strings to indexed data strings [9] or indexing data paths and data elements to efficiently check structural relationships. *In contrast*, in XML filtering, instead of the data (which is transitionary), the collection of filter patterns themselves need to be indexed.

Automata-based Schemes. The state of the art in XML filtering include, finite state automaton(FSA)-based schemes, YFilter [4, 13] XScan [18], and XQRL [15]. In these, each data node causes a state transition in the underlying finite state automata representation of the filters. The *active states* of the machine usually correspond to the prefix matches identified in the data. In general, for deep and recursive XML data, *the number of active states can be exponentially large* [7, 8, 13, 16]. In fact, [16] shows that for a linear filter, an eager deterministic FA has $O(num\_ancdec\_axes \times query\_depth \times alphabet\_size^{\frac{num\_wildcards}{num\_ancdec\_axes}})$ active states. When there are multiple path expressions, [16] shows that an eager DFA may result in $O(2^{num\_path\_expressions})$ active states. Furthermore, most of the states may never lead to results. [16] showed that using a lazy (as opposed to eager) scheme may reduce the number of active states to $O(query\_depth^{degree\_of\_recursion\_in\_data})$. YFilter uses a non-deterministic FA for reducing the number of automata states. However, since during runtime each NFA state can be visited (and inserted into runtime storage) multiple times, as stated in [13], deep documents could theoretically cause an exponential blow-up in the number of *active, run-time states*.

Push Down Approaches. XPush [17], instead, translates the collection of filter statements into a single deterministic pushdown automaton. The *eager* Xpush machine still needs exponential number of states. However, [17] also proposes a *lazy* implementation, which delays the discovery of the states and avoids redundant state enumeration, to bring the memory requirement down to $(\frac{1-\sigma^{k+1}}{1-\sigma})^n$, where n is the number of filter queries, σ is the selectivity of the predicates, and k is the number of predicates per query. SPEX [23] uses a network of transducers to evaluate regular path expressions with XPath-like qualifiers. For representing the transducer stack, SPEX needs memory quadratic in stream depth. Like SPEX, XSQ [24] uses a push-down transducer based approach for XPath filtering, where stacks keep track of matching begin and end tags and buffers store potential results to compute predicates.

Alternative Memory Organizations. PathM [12] uses a stack representation (where one stack is associated to each query no-

de) of data to obtain a memory requirement bound by the size of the query *times* the document depth. TurboXPath [19] avoids the translation of the query into a finite state machine and requires memory only linear in query size. [7] also deviates from automata and transducers to achieve $O(query\_size \times degree\_of\_recursion\_in\_data \times log(data\_depth))$ space and $O(query\_size \times data\_size \times degree\_of\_recursion\_in\_data)$ time per registered query. XTrie [10] represents path expressions as strings and indexes them into a trie structure, which leverage prefix commonalities in filters. The trie is used for detecting the occurrence of matching substrings as the input document is parsed. FiST [21] also represents queries as sequences; however, unlike XTries, these sequences represent each filter query wholistically and, thus, each query pattern is filtered independently without leveraging any prefix sharing. In fact, most schemes can not exploit both prefix and suffix commonalities. Yet, as experiments in Section 8 show, the best results are obtained when **both** prefix and suffix sharing are exploited simultaneously.

1.2 Contributions of this Paper

We first note that the execution time of any filtering scheme is lower bounded by the result enumeration step: since a given data path can result in exponential (in depth) number of matches[1], *major savings in execution time can only come from effective prefix and suffix sharing across filters.* However, sharing can be costly: if all matching prefixes are enumerated and stored naively as automata states, this may require extensive storage and time [7, 13, 16].

We note that a new and adaptive design, which can leverage different, hitherto conflicting, approaches simultaneously can help with both storage and execution time challenges. Such a mechanism should

- leverage *both prefix and suffix commonalities* across filter statements for reducing overall filtering time,
- *avoid unnecessarily eager result/state enumerations* (such as NFAs enumeration of active states), and
- *decouple memory management task from result enumeration* to ensure correct results even when memory is tight.

With these in mind, we introduce *AFilter* (Section 2), which encompasses the following advantages:

Simultaneous Prefix and Suffix Sharing: The proposed approach benefits from prefix commonalities across path expressions, while *simultaneously* leveraging suffix commonalities to reduce the cost of exploration of the potential matches.

Delayed State/Result Enumeration: As opposed to the rather *wasteful* active state enumeration mechanisms of finite-state-machine based approaches, AFilter uses a lazy mechanism to discover relevant matches only when an interesting *trigger* condition occurs. In the proposed scheme, the trigger conditions are associated with the leafs (last name tests) of query patterns to benefit from the generally more stringent data selectivities at the leaves.

Decoupling of Prefix-Caching (Efficiency) from Result Enumeration (Correctness): AFilter can function correctly with a runtime representation linear in filter and message depth. Query and data representations are discussed in Sections 3 and 4, respectively. Furthermore, AFilter can also leverage an on-demand prefix-caching mechanism (PRCache, Section 5), which can exploit additional memory to reduce filtering time, even if filter steps are clustered under common filter suffixes (Sections 6, 7).

Although the eventual goal of AFilter is to provide filtering for a large class of XQuery [1] statements, in this paper we focus on path expressions of type, $P^{\{/,//,*\}}$. Such path expressions are composed of query steps, each consisting of an axis (parent/child "/" or ancestor/descendant "//") test between data elements and a label test (including the "*"wildcard). In this respect, our approach is analogous to that of YFilter [4, 13], though we differ significantly in approach and provide significantly better performance, as evidenced in Section 8. The effective use of such path expressions in more complex scenarios, which could include predicates, twig queries of form $P^{\{[],//,*\}}$ [21], other types of axes [6], and more complete XQuery statements, have been discussed elsewhere [13, 14, 18, 20]. Therefore, within the context of these existing path expression based frameworks, in this paper, we only focus on the efficient and scalable filtering of the $P^{\{/,//,*\}}$ statements.

[1] Consider the extreme query "$//*//*//*$" and a data path of depth d. The total number of matches will be $O(d^3)$, i.e., exponential in the query depth.

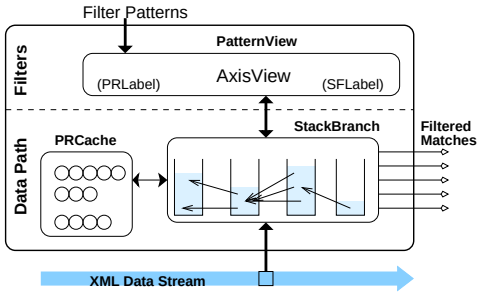


Figure 1: Overview of the AFilter architecture

2. AFILTER: PATH EXPRESSION FILTERING THROUGH PREFIX-CACHING AND SUFFIX CLUSTERING

AFilter is composed of three components (Figure 1): A linear size data structure, called *PatternView*, to index registered pattern expressions; a runtime data structure called *StackBranch* , (linear in the depth of the message tree) to represent the current data branch; and *PRCache*, which stores prefix sub-matches for re-use.

2.1 "PatternView" to Index Filter Patterns

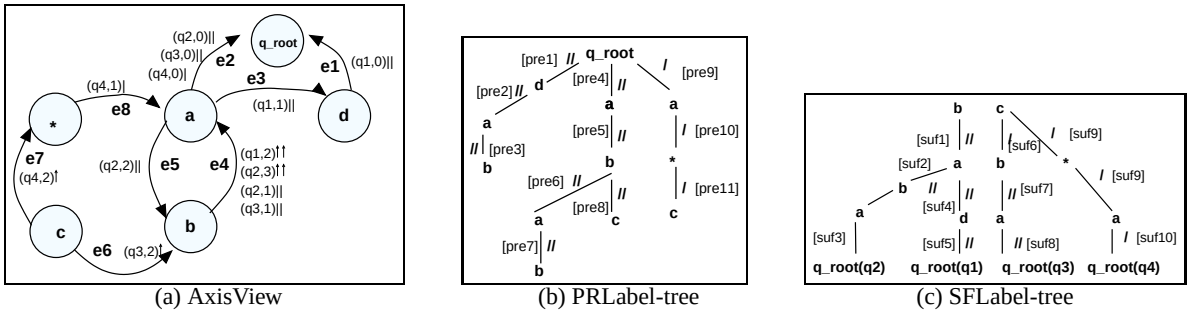
PatternView is a collection of linear size data structures that represent the query patterns that are registered in the system:

- **AxisView:** AxisView is a directed graph capturing the query steps registered in the system. Each node in the graph corresponds to a label and each edge corresponds to a set of axis tests. Each edge is annotated with a set of axis *assertions* that needs to be verified to identify matches.
- **PRLabel-tree:** PRLabel-tree is an (optional) "trie" data structure which clusters path expressions based on the commonalities in their prefixes. This is used for enabling prefix-based *sharing* of subresults across path expressions.
- **SFLabel-tree:** SFLabel-tree is an (optional) "trie" data structure which clusters path expressions based on their overlapping suffixes. It is used for clustering the evaluation of the *assertions* on the AxisView edges.

PatternView is incrementally maintainable. To complement the linear space PatternView data structure for registered filter expressions, we also use a compact stack-based representation (StackBranch) of the data being processed.

2.2 "StackBranch " for Encoding Data

StackBranch is a linear size data structure which represents the current root-to-element path being considered. StackBranch uses one stack per AxisView node. As the streaming data is consumed, the stacks are populated with *stack objects*. Pointers associated with these stack objects maintain the ancestor/descendant and



(a) AxisView (b) PRLabel-tree (c) SFLabel-tree

Figure 2: (a) AxisView for path expressions, $\{q_1 = //d//a//b, q_2 = //a//b//a//b, q_3 = //a//b/c, q_4 = /a/*/c\}$; (b) PRLabel-tree and (c) SFLabel-tree corresponding to the same queries

parent/child relationships across objects in stacks. StackBranch is used for identifying if there are any matches in the current branch when *trigger* conditions, associated with the leaves, are observed.

2.3 "PRCache" for Sharing Matches

The linear PatternView and StackBranch data structures overviewed above constitute the *base resources* needed for filtering of path expressions. However, when there is additional memory, AFilter can further exploit the overlaps in the prefixes of queries using a third, memory-adaptive, component.

PRCache is a cache structure for prefix-based sharing of subresults across path expressions. It leverages the clustering opportunities provided by the PRLabel-tree to eliminate redundant processing in AFilter. PRCache temporarily stores potential sub-matches; i.e., it is analogous to the concept of active states in finite state machine based schemes. However, *in AFilter, a path is materialized and cached only if it is included in at least one match.* Furthermore, since the correctness is independent of whether subresults are available in PRCache or not, the loosely-coupled structure (as opposed to hard coded data structures of existing filtering mechanisms) enables limiting the memory usage and provides opportunities for deployments in systems with bounded buffers.

3. PATTERNVIEW: A COMPACT REPRESENTATION OF FILTER EXPRESSIONS

PatternView is a linear space data structure for representing registered filter expressions.

3.1 AxisView: Axis-clustered Representation of Filters

A filter expression, of type $P^{\{/,//,*\}}$, is a sequence of steps where each step has a navigation axis (parent-child or descendent) and a label (name test) predicate. The AxisView data structure captures and clusters all axes of all filter expressions registered in the system in the form of a directed graph (Figure 2(a)).

Let $\mathcal{Q} = \{q_1, \ldots, q_m\}$ be a set of filter expressions. Let $\Sigma = \{\alpha_0, \alpha_1, \ldots, \alpha_r\}$, where $\alpha_0 =$ "*q_root*", be the label alphabet composed of the element names in the filter expressions in $\mathcal{Q}$. Let also $\Sigma_* = \Sigma \cup \{\alpha_*\}$ be the alphabet extended with the wildcard symbol, $\alpha_* =$ "$*$". The corresponding AxisView, $AV(\mathcal{Q}) = (V, E, AN)$, structure is a labeled directed graph:

- For each, $\alpha_k \in \Sigma_*$, V contains a node n_k.
- If there is an axis, "α_k/α_l" or "$\alpha_k//\alpha_l$", in filter predicates in $\mathcal{Q}$, then E contains an edge $\widehat{e}_h = \langle l, k\rangle$ from n_l to n_k.
- Each edge, $\widehat{e}_h$, has an associated annotation, $AN(\widehat{e}_h)$; each annotation contains a set of *assertions* that, if verified, can be used to identify a filter result.

  Let "α_k/α_l" or "$\alpha_k//\alpha_l$" be the s^{th} axis in a filter pattern, q_j. Furthermore, let $\widehat{e}_h = \langle l, k\rangle$ be the edge from n_l to n_k. Then, the set of annotations associated with $\widehat{e}_h$ contains an assertion $assert_h \in AN(\widehat{e}_h)$, such that

  - if the axis is of the form "α_k/α_l" then
    - if α_l is the last label test in the filter pattern, q_j, then $assert_h$ is "$(q_j, s)^{\uparrow}$" else $assert_h$ is "$(q_j, s)^{|}$"
  - if the axis is of the form "$\alpha_k//\alpha_l$" then
    - if α_l is the last label test in q_j, then $assert_h$ is "$(q_j, s)^{\uparrow\uparrow}$" else $assert_h$ is "$(q_j, s)^{||}$".

  The two symbols, $\uparrow$ and $\uparrow\uparrow$, in the assertions denote the trigger conditions (through parent/child and ancestor/descendent axes respectively).

EXAMPLE 1 (AXISVIEW EXAMPLE). *Consider the following four filter expressions:*

- $q_1 = //d//a//b$,
- $q_2 = //a//b//a//b$,
- $q_3 = //a//b/c$, *and*
- $q_4 = /a/*/c$.

Figure 2(a) illustrates the corresponding AxisView data structure.

Note that, unlike state machine-based schemes (such as YFilter [13]), AxisView is *not* an NFA traversed in a *forward* manner to generate partial matches. Instead, AxisView acts as a blueprint for the construction of the run-time data structure, StackBranch , which is traversed in the *reverse* direction and only when a *trigger* condition is observed. In fact, *if no trigger conditions are observed in the XML data stream, it is possible that no traversal will occur.*

3.2 Space Complexity of the AxisView Data Structure

If the size of $\mathcal{Q}$ is $size(\mathcal{Q})$, the size of $AV(\mathcal{Q})$ is also $size(\mathcal{Q})$; in other words, the AxisView data structure is linear in the size of filter statements. Furthermore, AxisView is incrementally maintainable. Details are omitted as both results are straightforward.

3.3 PRLabel-tree and SFLabel-tree

Query path expressions can also overlap in terms of their prefixes and suffixes. Most existing schemes rely on either prefix or suffix commonalities, but not both. AFilter, on the other hand, leverages both types of overlaps through (optional) linear *trie*-based data structures, PRLabel-tree and SFLabel-tree.

EXAMPLE 2 (PRLABEL-TREE AND SFLABEL-TREE). *Reconsider the expressions in Example 1. Figure 2 depicts the prefix and suffix labels computed using PRLabel- and SFLabel-trees.*

The use of PRLabel-tree for enabling prefix-based *sharing* of subresults across path expressions is discussed in detail Section 5. The use of SFLabel-tree for pruning unpromising trigger conditions is discussed in Section 6.

Note that, as long as the prefix and suffix labels are chosen such that they enable efficient discovery of parent/child and ancestor/descendant relationships in PRLabel-tree and SFLabel-tree, the actual data structures do not need to be maintained in the memory. Candidate labeling schemes for the PRLabel-tree and SFLabel-tree include the positional encoding or the extended Dewey [22].

4. STACKBRANCH FOR COMPACT DATA ENCODING

As its name implies, StackBranch uses a stack representation of the active XML data branch. In the literature, due to their compact representation of the XML data paths, stacks have been used to implement structural join operators on stored data. The relevant work includes TwigStack/ PathStack [9] and and StackTree-Desc/Anc [3]. More recently, stack-based schemes are also being considered for XML filtering schemes, such as XSQ [24], PathM [12], and XPush [17]. StackBranch leverages the AxisView graph discussed in the previous section to construct a highly compact representation of the runtime state of the data, suitable for both prefix- and suffix-clustered operations.

StackBranch , $SB(\mathcal{Q}) = \{S_k \mid \alpha_k \in \Sigma_*\}$, of $\mathcal{Q}$ is a set of stacks corresponding to the nodes of the AxisView, $AV(\mathcal{Q}) = (V, E, AN)$. StackBranch contains one stack for each node in the AxisView; i.e., *only one stack for each symbol in the label alphabet*. Stacks are also included for the root (q_root) and the "*" wildcard.

At any given point in time, *StackBranch represents the data path from the root of the current document to the last seen element.* Thus, as the streaming data is consumed (in a document-order manner), these stacks are populated appropriately with *stack objects*.

4.1 XML Message Stream

We use the conventional well-formed XML message model, where each message in the stream is an ordered tree of elements. The beginning of each element is marked with a *start tag* and its end is marked with an *end tag*; all the descendant elements start and end between these two tags. If x is an XML message, then $x[i]$ denotes the i^{th} element seen during the document-order (pre-order) traversal of x. The label, $\alpha_l = tag(x[i]) \in \Sigma$ denotes the label of this element and $depth(x[i])$ is its depth in the message. An XML stream is, then, a sequence $\{x_1, x_2, ...\}$ of XML messages.

4.2 Maintaining StackBranch

The runtime state of StackBranch is affected when a start tag of an XML element is encountered or when an end tag is seen.

Each time a start tag is observed in the data stream, a new *stack object* is created and is pushed into the stack corresponding to the element label. Each stack object contains the index of the element, its depth in the message, and as many pointers as the out-degree of the corresponding node in the AxisView data structure. Each pointer corresponds to an edge in the AxisView and points to the topmost object in the stack corresponding to the destination of the edge. If any of the queries also contain the "*" wildcard symbol, then for each new stack object inserted into its own stack, a corresponding stack object is created and inserted into the special S_* stack. The push algorithm is depicted in Figure 3.

EXAMPLE 3 (START TAG OBSERVED IN DATA).
Figure 4(a) shows the stacks of an empty StackBranch corresponding to the AxisView in Example 1 and Figure 2.

Push: (When a start tag, $< \alpha_l >$, for $x[i]$ is seen in the input stream)

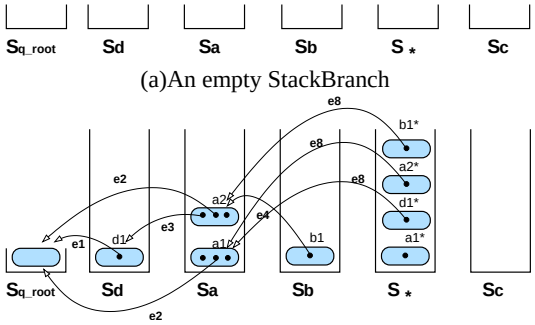
/* *Create a new stack object for the new element and push it into the corresponding stack ... Let t denote the number of outgoing edges of node n_l (corresponding to label α_l) in the AxisView data structure.*/

1. Create an object o of the form
$$o = \langle i, depth(x[i]), \langle ptr_1, .., ptr_t \rangle \rangle$$
2. If the h^{th} edge $\langle l, k \rangle$ of node n_l points to node n_k, then ptr_h will point to the topmost element of stack S_k. If S_k is empty, then $ptr_h = \perp$
3. push o into stack S_l.

/* *Create a new stack object for the new element and push it into the special "$*$" stack .....Let r denote the number of outgoing edges of the special node n_* corresponding to the $*$ wildcard*/

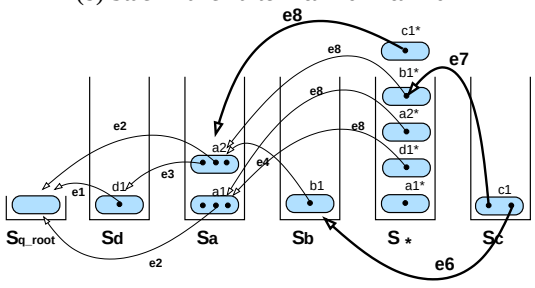
4. Create an object o of the form
$$o = \langle i, depth(x[i]), \langle ptr_1, .., ptr_r \rangle \rangle$$
5. If the h^{th} edge $\langle l, k \rangle$ of node n_* points to node n_k, then ptr_h will point to the topmost (non "i") element of stack S_k. If S_k is empty, then $ptr_h = \perp$
6. push o into stack S_*.

Figure 3: The push step is called each time an open tag is seen



(a)An empty StackBranch

(b) StackBranch after <a><d><a><b>



(c) StackBranch after <a><d><a><b><c>

Figure 4: (a) An empty StackBranch corresponding to the AxisView in Figure 2, (b) the status of the StackBranch after <a><d><a><b>, and (c) its status after <a><d><a><b><c>

There is one stack per label symbol (independent of the number of filter statements). Figure 4(b) shows the state of StackBranch after the stream <a><d><a><b> is observed. Figure 4(c), shows the StackBranch after the next tag, <c>, is observed in the stream.

When <c> is observed in the data, a new stack object c_1 is created an inserted into the S_c stack. This new stack object has two out-going pointers, corresponding to the edges $\widehat{e}_6$ and $\widehat{e}_7$ in the AxisView (Figure 2). These pointers point to the topmost objects of the destination stacks.

Pop: (When a stop tag, $< /\alpha_l >$, for $x[i]$ is seen in the input stream)

1. Remove and eliminate the topmost object in stack S_l.
2. Remove and eliminate the topmost object in stack S_*.

Figure 5: The pop step is executed each time a close tag is seen

Then, a new stack object, c_{1*} (again corresponding to <c>) is created and pushed into the S_* stack. This object has a pointer, corresponding to edge $\widehat{e}_8$, pointing the topmost object in S_a.

Note that, stack S_{q_root} always contains a single object. The special stack for "", on the other hand, contains one stack object for every element observed on the current root-to-node branch.*

As soon as the end tag of an element is seen, the corresponding stack object can be popped and eliminated (along with its pointers) from the data structure. The pop algorithm is shown in Figure 5.

EXAMPLE 4 (END TAG OBSERVED IN DATA). *If after seeing the data stream <a><d><a><b><c>, we encounter the end tag </c>, then, StackBranch reverts back to its state in Figure 4(b) from its state in Figure 4(c).*

4.2.1 Total Time Complexity of the Push and Pop

Each push operation has to create a new stack object and set all of its pointers to the topmost objects in the destination stacks. Therefore, given an XML message, x, and a set of filter expressions, Q, the worst case total cost of the *push* operations is $O(size(x) \times max\_fanout\_AV(\mathcal{Q}))$. Since the maximum number of outgoing edges in a StackBranch node is $|\Sigma_*|$ (i.e., $max\_fanout\_AV(\mathcal{Q}) \leq |\Sigma_*|$), the complexity of push is $O(size(x))$. The total cost of the pop operation is simply $O(size(x))$.

4.2.2 Space Complexity of StackBranch

The size of the StackBranch depends on how many stack objects are stored at a given point in time and the number of pointers associated with these objects:

- *Number of objects:* The total number of objects in the StackBranch at any point in time is $\leq 2d+1$ where, d is the depth of the last seen start tag. The factor of 2 comes from the fact that objects may need to be created for the S_* stack along with their own stacks.
- *Number of pointers:* For each object in the StackBranch , the maximum number of outgoing pointers is limited with the out-degree of the corresponding node in the AxisView. Again, the maximum number of outgoing edges is limited with the size of the label alphabet Σ_*. Therefore, the worst case number of pointers in StackBranch is bounded by $2d|\Sigma_*|$; i.e. it is independent of the number of queries.

In other words, since StackBranch associates only one stack per label symbol (as opposed to one stack for each filter query step) the memory requirement is linear in the message depth. In contrast, for instance, the memory requirement of PathM [12] is bound by the size of the query times the document depth.

4.3 TriggerCheck: Identifying Candidate Assertions

As mentioned in the introduction, as opposed to the finite automata based systems which traverse the state automata as they consume the input stream, AxisView and StackBranch structures are not traversed until a *trigger* condition is observed. To benefit from the generally more stringent selectivities in the leaves of XML data, we use the leaves of filter predicates as triggers. Thus, we consider the trigger *assertions* ($\uparrow$ or $\uparrow\uparrow$) associated with the AxisView edges corresponding to the pointers of the StackBranch objects. In particular, if the edge associated with a newly created pointer has a trigger assertion associated with it, then we know that the new stack object corresponds to the last node of at least one filter expression registered in the system. This is a trigger condition.

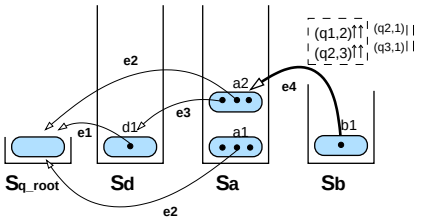


Figure 6: The <b> tag seen in the data triggers two assertions, $(q1,2)^{\uparrow\uparrow}$ for query $q_1 = //d//a//b$ and $(q2,3)^{\uparrow\uparrow}$ for $q_2 = //a//b//a//b$; only the relevant stacks in Figure 4 are shown in this figure

EXAMPLE 5 (CHECKING TRIGGER CONDITIONS). *In Figure 6, the stack object b_1 corresponding to the <b> open tag is pushed into the stack S_b. The AxisView edge $\widehat{e}_4$, corresponding to the outgoing pointer, has a total of four assertions: $(q1,2)^{\uparrow\uparrow}$, $(q2,3)^{\uparrow\uparrow}$, $(q2,1)^{||}$, and $(q3,1)^{||}$. Two of these, $(q1,2)^{\uparrow\uparrow}$ for filter $q_1 = //d//a//b$ and $(q2,3)^{\uparrow\uparrow}$ for $q_2 = //a//b//a//b$, are trigger assertions.*

Note also that, although the path expression $q_2 = //a//b//a//b$ has two bs, only the last (leaf) label test is triggered.

Once trigger assertions are identified, the system needs to verify whether these assertions correspond to any actual matches or not. In some cases, it is easy to deduce that trigger assertions are not promising. For instance, for a filter expression to have a match, there must be at least one pointer between all the relevant stacks. Also, the number of label tests in the filter query should be less than or equal to the depth of data. If these conditions do not hold, there can not be any matches. These pruning conditions can be implemented efficiently and can be useful, especially if the leaves have less stringent selectivities than earlier label tests in a given filter query. If an assertion is not pruned, then the StackBranch pointers have to be followed (or traversed) to identify whether there are actual matching path expressions.

Figure 7 shows the trigger phase operations. The processing of all non-pruned *candidate assertions* is performed by *traversing* the pointers outgoing from the triggering stack object (Step 3b). The traversal operation (discussed in Section 4.4) will return the subresults for all validated candidate assertions. These validated assertions will then be *expanded* by mapping with the matching subresults (Step 3c) and will be returned as results.

4.3.1 Time Complexity of the TriggerCheck Phase

Given an XML document, x, and a set of path expressions, Q, if we consider the extreme case where all insertions result in candidate assertions for all edges in the AxisView, then in the *worst-case*, there will be $O(size(x) \times size(\mathcal{Q}))$ *TriggerCheck* conditions which will need to be traversed for verification.

4.4 Pointer Traversal for Enumerating Matches

In this paper, we consider the general filtering problem, where the system returns all matching elements along the matched filter expressions (referred to a path-tuples in [14])[2]. Note that not

[2]This is not a strong requirement; more traditional XPath semantics, where only the element matching the last label test is returned, is a straightforward subset of the algorithms presented here.

TriggerCheck: (For an object $o = \langle i, depth(x[i]), \langle ptr_1, .., ptr_s \rangle\rangle$ pushed into stack S_l):

1. $tempresult = \emptyset$
2. $result = \emptyset$
3. For each ptr_h of o

/ Identify the candidate assertions*/*

(a) $cand =$
$\{(q,s)^{|} \mid (q,s)^{\uparrow} \in AN(\widehat{e}_h) \wedge \neg prune(q)\} \cup$
$\{(q,s)^{||} \mid (q,s)^{\uparrow\uparrow} \in AN(\widehat{e}_h) \wedge \neg prune(q)\}$

/ ...if $cand$ is not empty validate the candidate assertions by traversing the pointers*/*

(b) if $cand \neq \emptyset$ then
i. $tempresult = tempresult \cup$
$traverse(cand, depth(x[i]), ptr_h)$

/ Finally, merge the validated candidate assertions with the returned subresults. This step will be referred to as $expand(result, o, tempresult)$ in the rest of the paper*/*

(c) for all $r \in tempresult$
i. let $[(q,s), o_u]$ be the head of r
ii. $result = result \cup \langle [(q, s+1), o] | r \rangle$

4. Repeat the same process for o pushed into stack S_*

Figure 7: The TriggerCheck step is executed each time a new stack object is created

all candidate assertions triggered in the TriggerCheck step correspond to an actual match. Some triggered assertions, on the other hand, may correspond to multiple matches. Therefore, triggered assertions need to be verified and the corresponding matches must be identified. This process involves traversal of the pointers embedded in the StackBranch structure from the stack object, where a trigger condition is identified, back to the root object, q_{root}.

EXAMPLE 6 (TRAVERSING STACKBRANCH). *Figure 6 shows the two candidate assertions triggered due to the* <b> *tag seen in the data. Since the pointer associated with the corresponding stack object, b_1, points to the object a_2 in stack S_a, verification of this trigger will require the system to* traverse *the corresponding pointer towards the stack object a_2. Note that the pointer is traversed only once (in a grouped manner) for both candidates, $(q_1, 2)^{||}$ and $(q_2, 3)^{||}$, asserted by the trigger.*

The stack object a_2, on the other hand, has two outgoing pointers, one pointing to stack S_d and the other to S_{q_root}. These pointers are associated with AxisView edges, $\widehat{e}_3$ and $\widehat{e}_2$, respectively; therefore whether these pointers will be traversed or not depends on whether the local assertions associated with these two edges are compatible with the two candidate assertions, $(q_1, 2)^{||}$ and $(q_2, 3)^{||}$. We say that a candidate assertion $assert_i = (q_i, s_i)$ is compatible with a local assertion $assert_j = (q_j, s_j)$ if $q_i = q_j$ and $s_i = s_j + 1$.

Figure 8 illustrates the traversal process:

(a) First the pointer associated with edge $\widehat{e}_2$, from a_2 to q_root, is considered (Figure 8(a)). In this case, the only common filter query between the sets of candidate and local assertions is q_2. However, the required steps of the trigger assertion and the local assertion (2 and 0 respectively) do not match; naturally, a query step 2 can only be preceded by a query step 1. Therefore, this pointer does not lead into further traversals.

(b) When the outgoing pointer associated with edge $\widehat{e}_3$, from a_2 to d_1, is considered (Figure 8(b)), however, we find that there is a common query, q_1, which has matching assertions (steps 1 and 2, respectively). Therefore, there is a possible match and the outgoing pointers associated with d_1 should be further traversed.

(c) In the next step (Figure 8(c)), the outgoing edge, $\widehat{e}_1$ from d_1 is considered. In this case, the candidate $(q_1, 1)^{||}$ matches the local assertion $(q_1, 0)$. Therefore, the pointer can be traversed to its destination, q_root. Since the root is reached, this identifies a match for filter expression, q_1. The match consists of stack objects, d_1, a_2, and b_1.

(d) Since the stack based representation guarantees that any stack object under a_2 in the stack S_a will also be an ancestor of b_1 and since the candidate assertion being checked for $\widehat{e}_4$ is an ancestor/descendent axis, we need to look further down the stack to see if there are any further potential matches.

In this example, the S_a stack contains the stack object a_1 under a_2; therefore, a_1 needs to be considered. In this case, a_1 only has one single outgoing pointer, corresponding to the edge $\widehat{e}_2$; however, as before, the local filter conditions of $\widehat{e}_2$ do not match the incoming candidates. Hence, this object can not lead into further matches.

The *traverse* logic underlying this example is shown in Figure 9.

4.4.1 Time Complexity of the Traverse Operations

Since the number of candidate and local assertions to match against each other can be fairly large, a hash-join based scheme (which, for each incoming candidate assertion (q, s), searches for the hash of $(q, s-1)$ in the set of local assertions corresponding to a given pointer) is used for identifying the matches at Steps 7c and 7(e)v of the algorithm in Figure9; therefore, the cost of the matching of incoming and local assertions is linear in the number of assertions.

Some traversed candidate assertions result in matches, while others fail to return any (sub)results. Thus, the total cost of the traverse operations involves both. In the worst case (when the selectivities are less stringent at the leaves) the failing assertions will be recognized only when the traversal is already close to the query root object, q_{root}. Thus, the *worst case* complexities of both of these terms are similar. For a triggered candidate assertion, the number of traversals on StackBranch is bounded by $max\_query\_depth^{max\_stack\_depth}$. Note that $max\_stack\_depth \leq data\_depth$.

Since for lazy automata-based approaches [16], enumerating and storing each new state would cost time, there is a close correspondence between their time and space complexities. Note that, the above worst case time for AFilter is similar to the state enumeration complexity, $O(query\_depth^{degree\_of\_recursion\_in\_data})$, achieved by [16] using lazy DFA. [16] shows that this is significantly lower in complexity then the (theoretical) worst case of eager solutions.

In the case of AFilter, however, the above worst case complexity arises only when the basic, memoryless algorithm is used; i.e., when AFilter does not remember earlier traversals. Although this can be an advantage in low-memory installations, where the AFilter algorithm can work even when other algorithms may fail due to lack of sufficient memory, when there is extra memory, the filtering cost can be significantly reduced if positive (success) or negative (failure) results could be cached and reused.

5. PRCACHE: PREFIX-CACHING SUPPORT FOR ELIMINATING REDUNDANT POINTER TRAVERSALS

Let us reconsider Steps 7(d)i and 7(e)viA of the traverse algorithm in Figure 9. In these steps, for a given set ($cand[\widehat{e}_v]$) of candidate assertions, the corresponding pointer, ptr_v, is (recursively) traversed to verify the assertions and collect possible submatches.

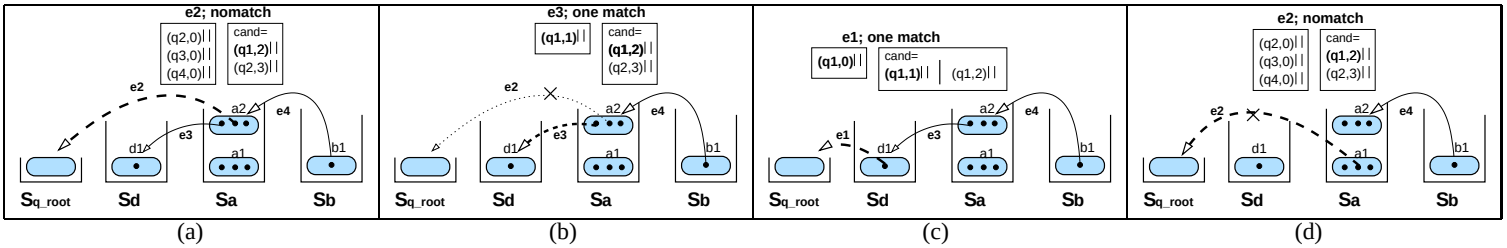


Figure 8: (a,b) Grouped verification of the candidate assertions associated with the two outgoing pointers of a_2, (c) a successful match, and (d) a no match case

As we stated earlier, for a given candidate assertion, traversal of a pointer can either be *successful*, i.e., can lead into one or more (sub)matches, or may *fail* to provide any results.

Note that if the same stack object is visited more than once during the filtering of an XML document (for example due to similar trigger conditions observed in the data), then it is possible that traversals originating from this object will repeatedly try to validate the same candidate assertions. This is wasteful: since stacks grow from root to the leaves in a monotonic fashion, it is straightforward to see that *for a given stack object, repeated evaluations of the same candidate assertion will always lead to the same result.*

Therefore, to avoid repeated traversals of the pointers in StackBranch for the same assertions, PRCache *caches* the success or failure of the candidate assertions associated with each traversed pointer (along with the results obtained during the first ever traversal of this pointer). This enables future traversals involving the same assertions to be resolved through an efficient table lookup.

5.1 Prefix-Caching for a Single Filter

Repeated traversals of the same step of the same filter expression is especially common in (a) tree structured data, where a shared portion of the data needs to be considered for multiple XML data branches or (b) in recursive data with repeated element names which can trigger the same filter multiple times.

Given a pointer, ptr, and an assertion, $assert$, associated with this pointer, PRCache caches the $traverseresult$, returned in Steps 7(d)i and 7(e)viA of the Traverse algorithm (in Figure 9) for the $\langle assert, ptr\rangle$ pair. Thus, next time the same assertion needs to be validated through the same pointer, the algorithm simply returns the corresponding matches from the PRCache; in other words, each prefix of each query is discovered only once.

This loosely-coupled memory structure enables AFilter to scale to the available memory space: unlike the existing mechanisms, if the cache storage space is limited, AFilter can completely eliminate the use of PRCache or can use cache replacement policies (such as LRU) to keep an upperbound on the number of cached prefixes, maximizing the utilization of the cache.

A second and (in terms of memory) cheaper caching alternative is to cache only the failed verifications (i.e., assertions with empty matches in $traverseresult$). In this approach, since the positive results are not cached, the same sub-matches may be identified multiple times. However, it eliminates repeated $fail\_traverses$ and since positive results are not cached, it has a significantly lower (linear in the number of query steps) cache storage demand.

5.2 Sharing Caches across Filter Expressions

A cached result for an assertion $assert_1 = (q_1, s_1)$ can be used for another assertion $assert_2 = (q_2, s_2)$, if we can ensure that $assert_1$ and $assert_2$ have identical intermediate results. In other words, prefix-commonalities across filter statements can be exploited for improving the utilization of the PRCache entries.

AFilter exploits prefix-commonalities through PRLabel-tree

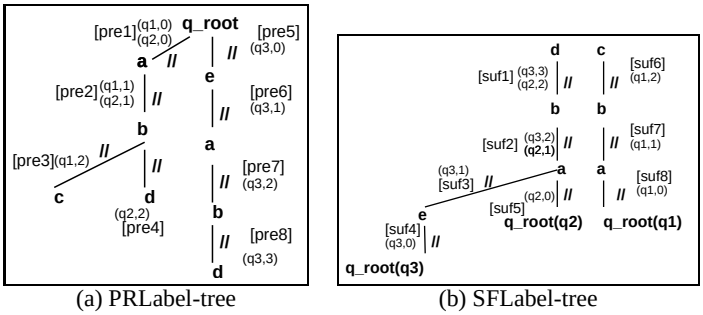


Figure 10: (a) PRLabel-tree and (b) SFLabel-tree for $q_1 = //a//b//c$, $q_2 = //a//b//d$, and $q_3 = //e//a//b//d$

(trie) data structure which labels common prefixes across path expressions. The entries in PRCache are then hashed in such a way that query steps sharing the same prefix also share cached results.

EXAMPLE 7 (PREFIX SHARING). *Consider*
- $q_1 = //a//b//c$,
- $q_2 = //a//b//d$, *and*
- $q_3 = //e//a//b//d$.

Figure 10(a) depicts the prefix clustering of individual query steps. Any assertions clustered under the same prefix ID in this figure can be cached under the same cache index. In this example, pairs, $(q_1, 0)$-$(q_2, 0)$ *and* $(q_1, 1)$-$(q_2, 1)$*, of assertions can be cached under the same prefixes,* pre_1 *and* pre_2*, respectively.*

6. SHARING WITH SUFFIX-COMPRESSED AXISVIEW

Prefix caching is useful in eliminating redundant traversals of the StackBranch pointers. However, even when such redundant traversals are eliminated, the cost of the Step7c of the traversal algorithm (Figure 9), where candidate assertions are matched against the local assertions associated with the outgoing pointers, can be high.

As discussed in Section 4.4, StackBranch implements this through a hash-join; thus, the cost of the operation is linear in the number of candidate assertions to be matched. Naturally, *reducing the number of candidate assertions would also reduce the time spent at the Step7c of Traversal*. Since traversals are from the leaves toward the root, clustering assertions in terms of shared suffixes would reduce the number of candidate assertions to consider.

EXAMPLE 8 (SUFFIX SHARING). *Let us consider*
- $q_1 = //a//b$,
- $q_2 = //a//b//a//b$, *and*
- $q_3 = //c//a//b$,

which all share a common suffix ($//a//b$*). The corresponding SFLabel-tree structure (shown in Figure 13(a)) captures this suffix overlap.*

Note that the original AxisView data structure, shown in Figure 13(b), does not capture the suffix commonality ($//a//b$*) across*

Traverse: $traverse(C, d, ptr)$

/ If no edges to traverse, return back*/*

1. if $ptr = \perp$ then return $\emptyset$
2. assume ptr points to $o' = \langle z, depth(x[z]), \langle ptr_1, .., ptr_u \rangle \rangle$

/ Eliminate those conditions that do not satisfy the parent/child conditions*/*

3. eliminate from C all $c = (q, s)^{|} \in C$ such that $depth(x[z]) \neq d - 1$
4. if $C = \emptyset$ then return $\emptyset$
5. $result = \emptyset$

/ If "q_root" is reached, then a match is found*/*

6. if α_z is "q_root" then
   (a) $result = C$
   (b) return $result$

/ Otherwise, for all outgoing pointers...*/*

7. for $v = 1$ to u do
   (a) let $\widehat{e}_v$ be the edge corresponding to the pointer ptr_v of o'
   (b) $cand[\widehat{e}_v] = \emptyset$

   / ..consider all incoming candidates to find matching local assertions (this step is implemented best using a hash-join)*/*

   (c) for $c_i \in \{(q_i, s_i)^{|} \in C\} \cup \{(q_i, s_i)^{||} \in C\}$
      i. for $c_j \in \{(q_j, s_j)^{|} \in AN(\widehat{e})\} \cup \{(q_j, s_j)^{||} \in AN(\widehat{e})\}$ such that $q_i == q_j$ then
         A. if $s_j == s_i - 1$ then $cand[\widehat{e}_v] = cand[\widehat{e}_v] \cup \{c_j\}$

   / ..if there are any matching local assertions, verify them by recursively traversing the corresponding pointer*/*

   (d) if $cand[\widehat{e}_v] \neq \emptyset$ then
      i. $traverseresult = traverse(cand[\widehat{e}_v], depth(x[z]), ptr_v)$
      ii. $result = expand(result, o', traverseresult)$

. . . */* Continued..*/*

.... . */* Continued..*/*

/ We need to also consider those stack objects that are below the current object; they may be relevant to the query*/*

(e) for all $o'' = \langle w, depth(x[w]), \langle ptr''_1, .., ptr''_u \rangle \rangle$ under o' in the stack
   i. let ptr''_v be the v^{th} pointer of o''
   ii. let $\widehat{e}_v$ be the edge corresponding to the pointer ptr''_v
   iii. $cand[\widehat{e}_v] = \emptyset$

   / ...objects further down the stack can not be parents; so, ignore parent/child assertions*/*

   iv. eliminate from C all $c = (q, s)^{|} \in C$
   v. for $c_i in \{(q_i, s_i)^{||} \in C\}$
      A. for $c_j \in \{(q_j, s_j)^{|} \in AN(\widehat{e})\} \cup \{(q_j, s_j)^{||} \in AN(\widehat{e})\}$ such that $q_i == q_j$ then
         - if $s_j == s_i - 1$ then $cand[\widehat{e}_v] = cand[\widehat{e}_v] \cup \{c_j\}$

   / if there are any matching local assertions, we need to verify them by recursively traversing the corresponding pointer*/*

   vi. if $cand[\widehat{e}_v] \neq \emptyset$ then
      A. $traverseresult = traverse(cand[\widehat{e}_v], depth(x[w]), ptr''_v)$
      B. $result = expand(result, o'', traverseresult)$

/ Return all the collected subresults along with the validated assertions*/*

8. return $result$

/ NOTE: This is a simplified representation of the traverse(). The actual implementation contains further optimizations....The join operations in Step 7c is implemented using a hash-join.... Also, Step 7(e)v is implemented in a way that benefits from the assertion matching already performed in Step 7c for earlier objects*/*

Figure 9: The Traverse step is called for verifying a set of candidate assertions; C is the set of candidate assertions, ptr is the pointer being followed, and d is the depth of the source stack object

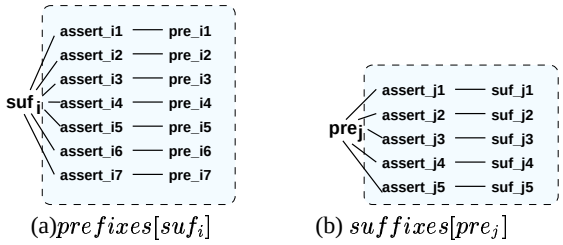
the three filter statements. The edge $\widehat{e}_4$ in the AxisView triggers each of these three queries independently. A suffix-compressed *AxisView reduces the amount of triggering and the traversals by clustering the shared suffixes in the AxisView, as shown in Figure 13(c). In this suffix-compressed AxisView example, there is only one trigger associated with edge $\widehat{e}_4$ which clusters all three queries.*

In the suffix-compressed AxisView, assertions are not made in terms of query IDs and steps, but in terms of edge IDs in the SFLabel-tree tree. The StackBranch is traversed towards the q_{root} in a suffix clustered manner: matching of the candidate assertions and the local assertions (to decide which pointers to traverse for which assertions) is performed by checking if two corresponding edges are neighbors in the SFLabel-tree tree or not.

Once the q_{root} is reached and the matches are being compiled by tracing the matching results back (Steps 7(d)ii, 7(e)viB of the traverse algorithm in Figure 9), the individual assertions clustered under the successful suffix labels are used to expand submatches to identify the individual results.

7. PREFIX-BASED CACHING WITH SUFFIX-COMPRESSION

A label in a suffix-compressed AxisView clusters suffixes of the filter patterns, whereas PRCache caches intermediary results based on the common prefixes of the filters. As shown in Figure 11, there may be many to many relationships between prefix and suffix



(a)$prefixes[suf_i]$ (b) $suffixes[pre_j]$

Figure 11: Many-to-many rel. between suffix-and-prefix labels

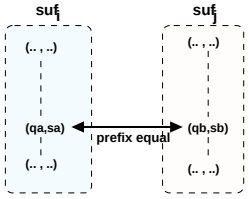
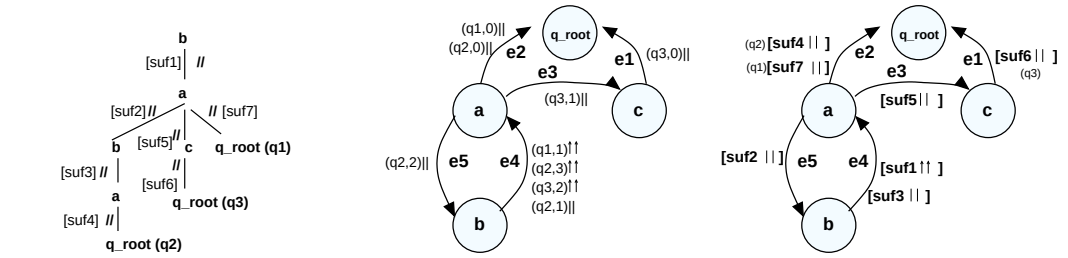


Figure 12: Although the two assertions, (qa, sa) and (qb, sb), share the same prefix, two different suffix label labels, suf_i and suf_j, place these two assertions into separate clusters. Thus, if in AxisView, assertions are clustered under suffixes, then these two assertions can not benefit from each others' prefix caches

labels. Unfortunately, suffixes and prefixes are not always compatible and suffix-based clustering can prevent prefix-based caching



(a) SFLabel-tree for suffix clustering (b) Original AxisView (c) Suffix-compressed AxisView

Figure 13: Suffix support for reduced filtering ($q_1 = //a//b$, $q_2 = //a//b//a//b$, and $q_3 = //c//a//b$)

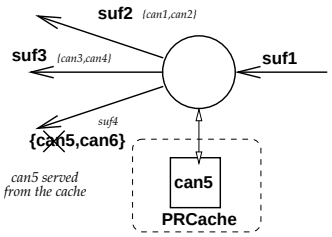


Figure 14: Early unfolding of suffix clusters: since $can5$ can be served from the cache, the corresponding cluster with the suffix label, suf_4, is unfolded; the corresponding pointer will be traversed in an unclustered manner (while the unaffected pointers continue to be traversed in a suffix-compressed manner with suffix labels, suf_2 and suf_3)

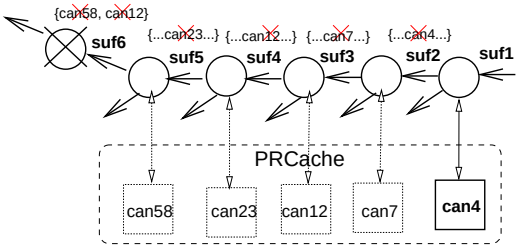


Figure 15: Late unfolding of suffix clusters, with candidate assertion removal and branch pruning

opportunities. In particular, some of the prefix commonalities in filter statements will be hidden by suffix labels (Figure 12). This naturally reduces the utilization rate of the cache.

EXAMPLE 9 (SUFFIX VS. PREFIX SHARING). *Let us reconsider the following three filter statements, first considered in Example 7 and Figure 10:*

- $q_1 = //a//b//c$,
- $q_2 = //a//b//d$, *and*
- $q_3 = //e//a//b//d$.

It is easy to see that the prefixes ($//a//b$) of filter statements q_1 and q_2 overlap, whereas the suffixes ($//a//b//d$) of q_2 and q_3 are also identical. The corresponding PRLabel-tree and SFLabel-tree data structures are shown in Figures 10(a) and 10(b), respectively.

This leads to a conflict: for prefix sharing, $(q_2, 1)$ needs to be able to access the cached results of $(q_1, 1)$; on the other hand, to benefit from suffix clustering, $(q_2, 1)$ needs to be clustered with $(q_3, 2)$, under the suffix label, suf_2.

Thus, benefiting from prefix caching, while also exploiting suffix clustering, requires unfolding (or unclustering) of suffix-based clusters as needed. There are two unfolding alternatives, *early* and *late* unfolding, both of which we discuss below.

7.1 Early Unfolding of Suffi x Clusters

During the backward traversal of the StackBranch , the early unfolding mechanism un-clusters a suffix-label as soon as the system determines that one of the candidate assertions contained in a suffix-based cluster can be delivered from the cache. Let us assume that, during the pointer traversal step, we identify that a candidate assertion, (q_j, s_j), clustered under the suffix label, suf_i, can benefit from a result already in PRCache. In early unfolding, the suffix label, suf_i, will be immediately unfolded and all the candidate assertions clustered under suf_i will be further verified individually.

EXAMPLE 10 (EARLY UNFOLDING). *Figure 14 illustrates this process with an example. If prefix caching is not used, the incoming suffix-label suf_1 will result in traversals of suffix labels, suf_2, suf_3, and suf_4, on three outgoing pointers.*

Let us assume that, the suffix label suf_4, clusters two candidate assertions, can_5 and can_6 and the assertion can_5 can be served from the cache. In this case, to benefit from the cached results, the early unfolding mechanism would stop traversing the pointer corresponding to suf_4 in the suffix domain. Instead, it would traverse the pointer for the individual non-cached assertion (can_6 in this example). The pointers that can not benefit from the cache will continue to be traversed in a suffix clustered manner (suf_2 and suf_3).

While PRLabel-tree and SFLabel-tree data structures are constructed, prefix IDs are associated with the suffix labels (Figure 11). When an assertion with a given prefix ID, pre_j is cached in PRCache, an $unfold[suf_i]$ bit for each suffix label, $suf_i \in suffixes[pre_j]$, is set (Figure 11(b)). If a suffix label with a set unfold bit needs to be traversed, that suffix label will be immediately unclustered and the individual assertions will be traversed independently.

7.2 Late Unfolding of Suffi x Clusters

Unfolding has an associated cost in terms of the lost assertion clustering opportunities. Especially in cases where (a) suffix clusters are large, but (b) the prefix cache hit rate is low (i.e., when only a few candidate assertions per suffix cluster can actually be served from the prefix cache), early unfolding can cause unnecessary performance degradations. In such cases, it may be more advantageous to *delay* the unfolding of suffix-clusters.

EXAMPLE 11 (LATE UNFOLDING). *Consider Figure 15, where the suffix label suf_2 contains a candidate assertion, can_4, which can be served from the cache early in the traversal process.*

In this case, early unfolding would require unfolding of all the assertions clustered under the label suf_2. However, if suf_2 is unfolded at this stage, none of the subsequent steps can be performed in the suffix clustered domain.

In contrast, late unfolding refrains from immediately unclustering the set of candidate assertions under suf_2. While can_4 is served locally from the cache, the edge corresponding to the suffix-based label suf_2 continues to be traversed using the suffix label, instead of being traversed as individual assertions.

The challenge with such a *delayed (or late) unfolding* mechanism, however, is to ensure that cluster domain traversal does not cause *redundant* work for the already cached result. In the above example, since can_4 will eventually be served from the cache, this assertion should be removed from further consideration to prevent redundant work: in other words, the semantics of the suffix-label, suf_2, needs to be modified to exclude can_4 (illustrated with a cross on can_4 in Figure 15). Thus, when an assertion with a given prefix ID, pre_j is cached in PRCache, a $remove[suf_i][pre_j]$ bit is set for each suffix label, $suf_i \in suffixes[pre_j]$ (Figure 11(b) illustrates $suffixes[pre_j]$).

7.2.1 Pruning Redundant Prefix Cache Accesses

If an assertion can be served from the cache, its prefixes do not need to be served from their own caches. Therefore, if an assertion is marked for removal from its suffix cluster, its prefixes should also be removed from their corresponding suffix-labels.

EXAMPLE 12 (PRUNING CACHE ACCESSES). *This is illustrated in Figure 15: let us assume that candidate assertions can_7, can_{12}, can_{23}, and can_{58}, are all prefixes of the candidate assertion can_4, which is removed from consideration. The cache will not be accessed for such non-maximal prefixes.*

In AFilter, when a suffix label, suf_i is traversed, each pre_j such that $remove[suf_i][pre_j]$ is being set, is also inserted into a prune set. This is achieved by setting a $prunecache[pre_j]$ bit. Note that, if pre_k is a prefix of pre_j, then $remove[suf_i][pre_j] \rightarrow remove[suf_i][pre_k]$, and

$$prunecache[pre_j] \rightarrow prunecache[pre_k].$$

This is used for pruning non-maximal prefixes of removed prefix labels from further consideration.

7.2.2 Pruning Redundant Traversals

Under late unfolding, if all candidates clustered under a suffix label are *removed* (i.e., can be served from the cache), the corresponding pointer does not need to be further traversed.

EXAMPLE 13 (PRUNING TRAVERSALS). *In Figure 15, suf_6 clusters only two candidate assertions, can_{58} and can_{12}, both of which have been marked for removal from consideration. Therefore, the corresponding pointer does not need to be further traversed.*

Pruning condition for suffix, suf_j, is checked by considering whether $\forall pre_j \in prefixes[suf_i]$ the removal bit $remove[suf_i][pre_j]$ is set or not (Figure 11(a) illustrates $prefixes[suf_i]$).

The performance of late unfolding depends on how easy it is to look into the clusters for checking (a) if any of the clustered assertions can be served from the cache, (b) if any of such assertions are in the removal list, or (c) if each candidate clustered under a suffix label is a prefix of another one which has already been removed.

We note that the cost of checking if any of the clustered assertions can be served from the cache is the same an early unfolding scheme would have to pay to use the PRCache. On the other hand, as described above, sharing of the removal bits between prefixes requires the propagation of the removed prefixes along the traversal path using the $prunecache[prefixID]$ bits. As evidenced in the next section, despite this overhead, late unfolding provides the best of both *prefix caching* and *suffix-clustering* approaches, and thus significantly outperforms all alternatives.

Acronym	Filtering approach
YF	YFilter
AF-nc-ns	AFilter, no cache, no suffix compression
AF-nc-suf	Suffix Compressed AFilter, no cache
AF-pre-ns	AFilter, prefix caching only, no suffix compression
AF-pre-suf-early	Suffix Compressed AFilter, prefix cache, early unfolding
AF-pre-suf-late	Suffix Compressed AFilter, prefix cache, late unfolding

Table 1: Notation used for various filtering deployments

Parameter	Values
Number of filter statements	10K-100K
XML message depth	~ 9
Average XML filter depth	~ 7
Maximum XML filter depth	15
XML message size	6000 bytes

Table 2: Experiment parameters (unless specified otherwise)

8. EXPERIMENTAL EVALUATION

In the previous sections, we discussed the various components of the AFilter algorithm for efficient and adaptive filtering of path expressions. In this section, we provide an extensive evaluation of AFilter through comparisons of the various properties of AFilter with those of a state-of-the-art XML filtering algorithm, YFilter [13]. We chose YFilter for comparison, as both AFilter and YFilter primarily perform filtering of path expressions. Unlike AFilter, however, YFilter relies on a finite state automata based approach and requires maintenance of all active states in the memory. Furthermore, YFilter exploits only prefix commonalities between filter statements, while AFilter can exploit both. Therefore, YFilter provides opportunities for a one-to-one comparison of various novel approaches underlying AFilter. Table 1 provides an overview of different filtering setups, with various AFilter components are turned on and off, we compared against YFilter.

We have implemented AFilter in Java (JDK 1.5). For comparison purposes, we used the YFilter implementation available at [2]. All experiments were conducted on a 1.7GHZ Pentium 4 machine with 1GB RAM. For the experiments reported in this section, unless stated otherwise, we generated XML data using the NITF DTD available through the YFilter test suites [2] and the ToXgene data generator [5]. The filter queries were generated using YFilter's query generator. Table 2 lists the various parameters and parameter value ranges we used for generating the data and filter queries we are reporting in this section. We also experimented with different parameters (such as *query/data depth, message size, and skewness*); results were consistent with the sample we are reporting.

8.1 Time vs. Number of Filter Expressions

Figure 16 shows the effect of the number of expressions on the time required for filtering a given set of filter expressions. For this experiment, we varied the number of filter expressions from 10K to 100K and we observed the filtering performance of different schemes. AFilter's performance varies depending on the scheme chosen. As expected the low memory AxisView-only (no-caching, no-suffix clustering) scheme takes more time than the others, including YFilter. Path-caching-only AFilter gives comparable results to YFilter, especially for smaller filter sets. On the other hand, when both suffix and path caching are used, the performance of AFilter becomes significantly better than YFilter, requiring only less than 15-30% of YFilter for large filter sets.

8.2 Comparison of Compression Approaches

Figure 17 compares the three suffix-compressed approaches to AFilter: suffix-compressed AxisView with no prefix caching (AF-nc-suf), suffix-compressed AxisView with prefix caching and early unfolding (AF-pre-suf-early), and suffix-compressed AxisView

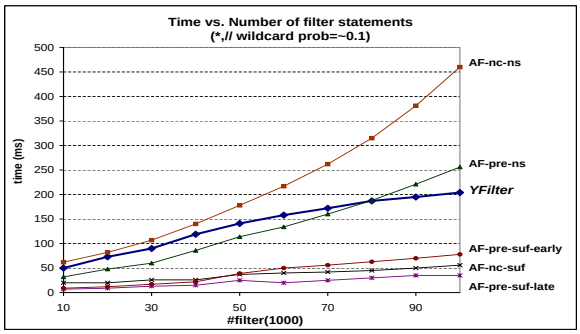


Figure 16: AFilter with suffix-clustering + prefix caching (with late unfolding) provides the best performance

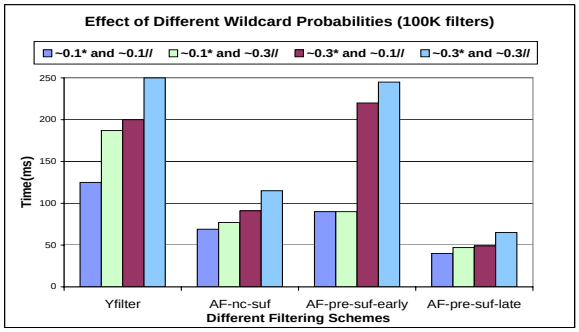


Figure 18: The impact of different wildcard compositions on filtering performance

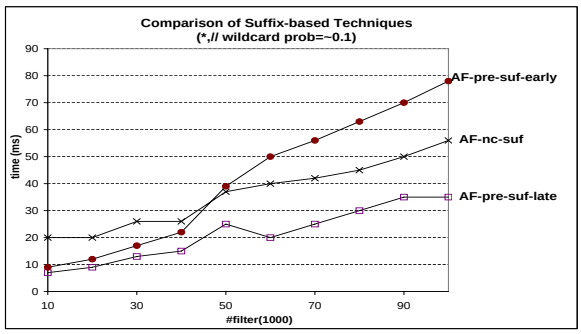


Figure 17: Comparison of different suffix-based approaches

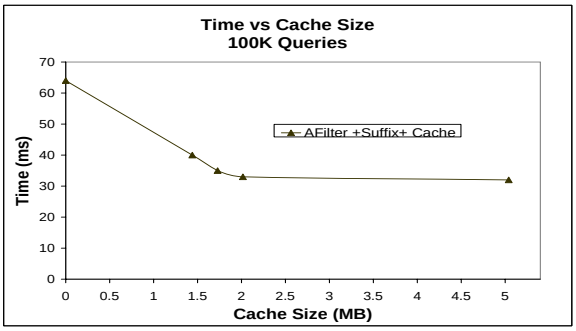


Figure 19: Impact of cache size on AFilter performance

with prefix caching and late unfolding (AF-pre-suf-late). As expected, when the number of filter statements are large, the loss in the suffix clustering opportunities due to early unfolding renders the AF-pre-suf-early scheme worse among the three. On the other hand, caching with late-unfolding (AF-pre-suf-late) brings together the desirable properties of both prefix caching and suffix-clustering, and performs significantly better than all other alternatives.

8.3 Time vs. Probability of Wildcards

Figure 18 shows the effects of different types and probabilities of wildcards on the various filtering schemes. As can be seen in this figure, both "*" and "//" affect the performance of YFilter. In contrast, the suffix-compressed AFilter schemes are less affected by the increase in the number of wildcards in the queries. One exception is the early-unfolding approach which is affected by the increase in the '*" wildcards; however, as expected, the suffix-compressed AFilter with prefix caching and late unfolding is minimally affected by the increase and outperforms all other alternatives.

8.4 Cache size vs. Time

Figure 19 shows that AFilter does indeed benefit from larger cache space, when available. As expected increasing the cache size improves AFilter's performance. Naturally, beyond some point, having more cache space does not help.

8.5 Number of Filters vs. Index Space

Figure 20(a) compares the amount of base memory requirement (i.e., AxisView) against the memory requirement of YFilter. As illustrated here, the base version of AFilter can run with lower available index memory than YFilter. Note that, for this dataset (since the number of unique labels are large and the depth of the data is relatively smaller), the index memory requirement significantly dominated the runtime memory requirement for both YFilter and StackBranch (Figure 20(b)).

8.6 Results for a Different Data Set

In order to verify that the above results hold across different data characteristics, we also experimented with a different DTD. For these experiments, we used the book DTD available at [1]. This DTD has a higher recursion rate and a smaller number of unique labels. In order to see the performance of different schemes in light- and heavy-usage usage of wildcards, we experimented with different wildcard ("*" and "//") occurance probabilities and different filter set sizes . Note that the numbers of distinct path expressions (of a given depth) generated under these scenarios are smaller since there are fewer unique labels.

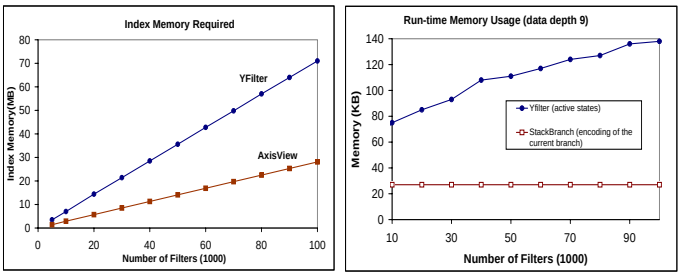
Figure 21 shows the results of YFilter against the suffix-compressed AFilter schemes (as shown before, without suffix clustering the runtime of AFilter is worse than YFilter). Suffix-clustering improves the performance. Once again, suffix-clustering with prefix-caching and late-unfolding outperforms other alternatives and consistenly requires less than 50% of YFilter.

9. CONCLUSIONS

In this paper, we introduced, *AFilter*, for filtering path expressions. In addition to the overview in Section 1.1, here we provide a short summary of how AFilter compares with the earlier work.

Automata-based schemes, such as YFilter, consume input symbols and traverse a state space. Active states have to be identified and maintained in an internal storage before the next symbol is considered. *The number of active states that have to be maintained in memory for the automata-based schemes can be exponentially large [7,13,16]*. In contrast, AFilter operates in two stages:

(a) As each symbol is consumed, AFilter constructs an interme-



(a) (b)

Figure 20: (a) Base memory needed to index queries with YFilter and AxisView and (b) memory needed to index run-time state of YFilter and StackBranch (this excludes cache space)

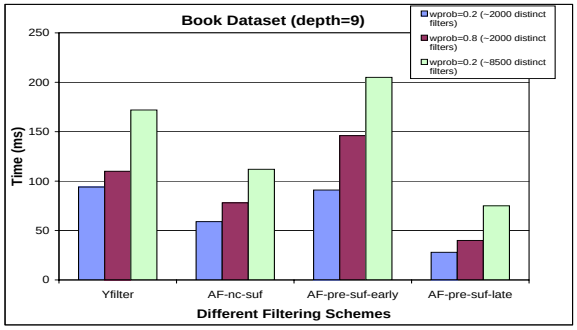


Figure 21: Filtering performances of the suffix supported AFilter schemes against YFilter for a different dataset.

diary StackBranch data structure. *StackBranch is a* linear *encoding of the XML data path.* Unlike YFilter's state storage, StackBranch does not contain any explicitly enumerated states or prefix matches.

(b) When a trigger condition is observed, the path-encoding pointers in StackBranch are traversed *backward* to stitch the individual "query steps" and to find and unfold path matches.

The stack-based linear data storage mechanism is analogous to the data indexing schemes used in TwigStack/ PathStack [9] and more recently in PathM [12]. However, unlike these schemes, StackBranch leverages the AxisView graph to construct a *highly compact representation of the run-time state of the data, when there are multiple filter queries (with prefix and suffix-overlaps).*

A primary advantage of AFilter is that when memory is tight, the pointers of StackBranch can be followed one-at-a-time, without having to allocate more-than-linear memory to store prefix matches. Thus, *AFilter provides tradeoff between memory and performance and can work with only linear memory*, when needed. In fact, AFilter uses an on-demand prefix caching mechanism (PRCache) which can exploit additional memory when available and can selectively cache prefixes. This triggering-initiated, lazy-enumeration scheme also benefits significantly from the more stringent selectivities of leaves (typical in practice).

Furthermore, unlike automata-based solutions (and many others) which are suitable only for prefix sharing, AFilter exploits simultaneously various sharing opportunities: common steps (AxisView), common prefixes (PRLabel-tree), and common suffixes (SFLabel-tree). Tries are fundamental data structures, used in many works (including YFilter [13] and XTries [10]) where prefix sharing is needed. Unlike most schemes (such as XTries) which use tries for matching path-segments, AFilter uses the PRLabel-tree/ SFLabel-tree to generate prefix and suffix labels that are used to annotate AxisView edges. Thus, *both prefix- and suffix-sharing can be exploited simultaneously* leading to significantly higher savings than is achieved by relying only one alternative (Section 8).

The experiment results in Section 8 showed that AFilter brings together the desirable properties of effective memory utilization, trigger-based result enumeration, and prefix-caching/suffix-clustering to enable scalable and high performance path expression filtering. In fact, the best results are obtained when both prefix and suffix clustering are exploited simultaneously.

10. REFERENCES

[1] XQuery 1.0: An XML Query Language.

[2] YFilter 1.0 release, http://yfi lter.cs.berkeley.edu/code. release.htm

[3] S. Al-Khalifa, H.V. Jagadish, J.M. Patel, Y. Wu, N. Koudas, and D. Srivastava. Structural joins: a primitive for effi cient XML query pattern matching. *ICDE* 2002

[4] M. Altinel and M. Franklin. Effi cient fi ltering of XML documents for selective dissemination of information. *VLDB* 2000

[5] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons. ToXgene: a template-based data generator for XML. *SIGMOD* 2002.

[6] C. Barton, P. Charles, D. Goyal, M. Raghavachari, M. Fontoura, and V. Josifovski. Streaming XPath processing with forward and backward axes. *ICDE* 2003.

[7] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over XML streams. *PODS* 2005

[8] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of evaluating XPath queries over XML streams *PODS* 2004

[9] N. Bruno, D. Srivastava, and N. Koudas. Holistic twig joins: Optimal XML pattern matching. *SIGMOD*, 2002.

[10] C. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Effi cient fi ltering of XML documents with XPath expressions. *ICDE* 2002.

[11] J.Chen, D.J.DeWitt, F.Tian, and Y.Wang. NiagaraCQ: A scalable continuous query system for internet databases. *SIGMOD* 2000.

[12] Y. Chen, S. Davidson, and Y. Zheng. An effi cient XPath query processor for XML streams. *ICDE* 2006.

[13] Y. Diao, M. Altinel, M.J. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *TODS*, 28(4):467-516, 2003.

[14] Y. Diao and M. Franklin. Query processing for high-volume XML message brokering. *VLDB* 2003.

[15] D. Florescu, *et al.* The BEA/XQRL streaming XQuery processor. *VLDB* 2003.

[16] T.J.Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream index. *TODS* 29(4):752-788, 2004.

[17] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. *SIGMOD* 2003.

[18] Z. Ives, A. Halevy, and D. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11 (4): 380–402, 2002.

[19] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *VLDB Journal*, 14 (2), pp. 197-210, 2005.

[20] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. *VLDB* 2004.

[21] J. Kwon, P. Rao, B. Moon, and S. Lee. Fist: scalable XML document fi ltering by sequencing twig patterns. *VLDB* 2005.

[22] J.Lu, T.W. Ling, C.-Y. Chan, and T. Chen. From region encoding to extended Dewey: on effi cient processing of XML twig pattern matching. *VLDB* 2005

[23] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifi ers against XML streams. *ICDE* 2003.

[24] F. Peng and S. S. Chawathe. XPath queries on streaming data. *SIGMOD* 2003.

[25] F. Peng and S.S. Chawathe, XPASS, http://www.cs.umd.edu/projects/xpass/index.html

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