Optimal Buffering for Streaming XPath Evaluation

Feng Peng
Department of Computer Science
University of Maryland, College Park
Maryland, USA, 20742
pengfeng@cs.umd.edu

Sudarshan S. Chawathe
Department of Computer Science
University of Maryland, College Park
Maryland, USA, 20742
chaw@cs.umd.edu

ABSTRACT

We motivate and present a definition of optimal buffering for streaming evaluation of XPath queries. We consider a large fragment of XPath that includes multiple (correlated) subqueries and reverse (up the document tree) axes. We describe a method for XPath evaluation with optimal buffering. We present the results of an experimental evaluation of our methods based on our implementation, which is freely available.

1. INTRODUCTION

XPath [5] is an important query language for XML, which is now used as a standard data format in a wide range of application such as data integration and information dissemination. XPath is a succinct yet powerful path language that can used to address parts of the XML documents. In most database systems that support XML, XPath is supported either as a standalone query language such as in the native XML databases [32, 16], or used together with SQL such as in the XML-extended relational databases [27, 17]. Moreover, it is a key component of other higher level XML query or transformation languages such as XQuery [4] and XSLT [31]. These languages usually use XPath to select a subset of the document tree and apply higher level operations, such as transformation rules and joins, on the selected subset. As XPath being such an important component of these database systems and languages, efficient evaluation method of XPath queries will benefit their performance.

An XPath query consists of a path expression called the *location path* that contains a sequence of *location steps*. For example, the following query returns the authors who wrote a book about XML:
`/descendant-or-self::book[subject="XML"]/child::author`.
There are two location steps in the query which are separated by "/": `descendant-or-self::book[subject="XML"]` and `child::author`. Each location step contains an *axis* before the "::" which specifies the relation between the current element and elements selected by previous location step, e.g., the `author` element should be a child of the `book` element in the previous location step, a *nodetest* which specifies the name of the element that is selected by this step, and an optional *predicate* such as `child::subject="XML"` that is enclosed in the brackets and specifies the conditions the selected elements have to satisfy. In XPath's abbreviated form in which the default axis is the child axis and the `descendant-or-self` axis can replaced by "//", the above query becomes much shorter: `//book[subject="XML"]/author`.

Although its form looks like the regular expressions, XPath is more powerful since it can use arbitrary XPath queries in the predicate and boolean operators to connect those subqueries. Moreover, XPath provides a set of thirteen axes that allows the query specify almost arbitrary pattern in the document tree. Since the predicates could be themselves complex patterns, traditional pattern matching algorithms can hardly be applied in XPath evaluation. Only recently did a polynomial evaluation algorithm for XPath is proposed in [10], which separates the path query into pairs of tags connected by *parent-child* or *ancestor-descendant* relations and joins the intermediate results after the pairs are retrieved from the data.

However, current evaluation methods are very inefficient when evaluating reverse axes such as **parent** and **ancestor**, as we will illustrated in the experimental results. The fundamental difficulty caused by the reverse axes is that they incur bottom-up traversal in the document tree while the tree is usually traversed in pre-order. In streaming environment, since seeking-back in the stream is usually not allowed or very expensive, this difference seems to be irreconcilable, or even claimed by some researcher "impossible".

Nevertheless, reverse axes are important for the users. Firstly, it empower the user to specify more complex patterns. Without reverse axes, XPath queries can only specify tree patterns, while with reverse axes the pattern could be a graph instead of tree. The details are explained further in this paper and a simple example here will show the difference.

EXAMPLE 1. *Some queries cannot be easily specified without reverse axes. Suppose we have a book dataset in which books are grouped by the publishers, which means that the book elements are children of the publisher elements. If we want to find a book that is either about XML or is published by O'Reilly, we have to use the* `parent` *axis:*
`//book[subject="XML" or parent::pub="O'Reilly"]`. *For*

such queries that the ancestor is optional, reverse axes are needed.

The reverse axes are also very convenient for user to specify query that can fit data of various DTDs or schema. For example, in the scenario of information dissemination, it is very likely a query issued by the user will be applied to heterogeneous data sources.

EXAMPLE 2. *It is very likely different book datasets organize the books differently. Suppose we want to find the book that is written by W3C and published by O'Reilly. However, the books can be grouped by the publisher, by the author, by both (in either order), or not grouped at all in which case the author and the publisher are children of the book instead of ancestors. With reverse axes, we can issue this query as:* `//book[pub="O'Reilly" or author="W3C" or ancestor::pub="O'Reilly" or ancestor::author="W3C"]`.

Moreover, some queries are more naturally composed using reverse axes. Since it is always the last location step that specifies the elements desired by the user, it is sometime more naturally think the ancestors as the prerequisites of the desired elements, which can be specified in the predicate instead of in the location path.

EXAMPLE 3. *In natural language processing, it is usually more straightforward to specify a child element in a parse tree of a sentence and refer to its ancestors. For example, the following query asks for the noun phrase that has a noun "book" and appears in a top level verb phrase (whose parent is the root of parse tree of the sentence) and contains a verb "read" and:* `//NP[ancestor::VP[parent::root and //V=read] and //N=book]` *(Here N is for noun, NP for noun phrase, and etc.) This query can be written without reverse axes, but its semantics are retained more explicitly using the current form.*

If the user cannot use reverse axes directly in the XPath query, the user has to either specify equivalent queries without these features or program on the intermediate result to get the final result. These approaches usually cause more management overhead and hard to maintain afterward. There are methods proposed such as in [26] to rewrite the XPath queries with reverse axes into equivalent queries without reverse queries. However, the method will either generate equivalent queries with join operations, which are difficult to evaluate (especially in streaming environment), or generate exponential number of queries (connected by disjunctions) with respect to the size of the query.

A solution to process the reverse axes in streams is to buffer all the data needed to evaluate the query and postpone the evaluation, which is used in the XAOS system [3]. There are three limitations of this approach. First, in the case of infinite stream, the evaluation may be unnecessarily postponed infinitely. Second, even if the predicate for an element is already evaluated, the method will still buffer the elements that are used in the predicate. Third, even the results of the predicates are all known, the correct result will not be sent to output and the items that has been proved not in the result are still buffered. Because of these limitations, this type of solution is inefficient and not viable for environments that need to process infinite streams or desire eager output.

We addressed the problem of streaming evaluation of XPath with predicates and closure axes in [28]. As we show in [28], multiple predicates and closure axes make it non-trivial to evaluate XPath queries over streaming XML data. As we tried to extend the work to handle reverse axes, the "irreconcilable" difference between the semantics of the reverse axes and the restrict of pre-order traversal made it difficult to apply the HPDT-based approach in the presence of reverse axes. An important assumption used in the HPDT architecture is that every predicate can be determined before the end of the element, which no longer holds with reverse axes in the predicate. However, after analyze the semantics of the more complex XPath queries with reverse axes, we found that it is still possible to process them in the streaming environment, which will be presented in this paper.

In this paper we propose a new method to evaluate XPath queries with reverse axes in the streaming environment. Based on a novel rewriting method that rewrites any XPath query into an equivalent single step XPath query and a compact dependency graph that encodes the relations among the undetermined elements for the query, the new method is proved to be efficient for both queries with and without reverse axes. Moreover, it needs only one pass of the data, even if there are arbitrary number of reverse axes in the query. Another desirable feature, especially useful in streaming environment, is that the results are emitted as soon as they are available. The memory usage of our approach is optimal since it buffers only the undecided potential result items. When evaluating complex predicates that may contain arbitrary subqueries connected by boolean operators, the algorithm terminates the evaluation of the predicate at the time the result is determined.

We may summarize our main contributions in this paper as follows:

- We define *optimum buffering* for streaming evaluation of XPath queries. Although the basic idea is simple, XPath features such as subqueries and reverse axes make a precise definition non-obvious. We note that it is very easy to generate query-stream pairs that require an unbounded buffer. Thus, we cannot hope to design algorithms with any fixed constraint on the buffer size.[1] Thus, this definition provides a guide for designing space-efficient algorithms for streaming XPath evaluation.
- We describe an efficient method for the streaming evaluation of XPath queries with optimum buffering. Our method supports a large fragment of XPath, including features such as multiple subqueries and reverse axes. (The main feature not included in this fragment is positional and sibling axes.) We provide a detailed justification of our claim of optimum buffering.

[1]We do not consider approximate query evaluation in this paper.

- We describe the implementation of our method in the XSQ streaming XML engine. Our implementation is freely available at `http://www.cs.umd.edu/projects/xsq/`.
- We present an experimental evaluation of our methods. Our results suggest that the analytical properties of our method map well to practical benefits.

The paper is organized as follows. In Section 2, we introduce the XPath language and the SAX model used in the paper. We then define the problem of optimal buffering in Section 3. Our approach and the algorithms are then introduced in Section 4. The correctness of the approach is briefly discussed in Section 5. Some implementation related discussion is presented in Section 6. In Section 7 we discuss some of the related works. The experimental results are illustrated in Section 8.

2. PRELIMINARIES

2.1 XPath

XPath queries consists of a sequence of **location steps** of form `a::n[p]` where a is an **axis** whose value can be one of {`self,child, descendant, descendant-or-self, parent, ancestor, ancestor-or-self`} in this paper, n is a **nodetest** that specifies the tag of the elements that may match this location step, and p is an optional **predicate** that specifies the condition the element has to satisfy to match this location step. The condition can be a single XPath query, which we call a **subquery**, or several subqueries connected using boolean operators (*and*, *or*, and *not*).

An axis is always evaluated on a *context*. The context contains an element, together with the size of the ordered result set where the element resides and the position of the element in the result set, If we do not refer the position of the element in the query, which is the case in the queries considered in this paper, we can ignore the size and position components if the context. Therefore, we can consider that an axis is always evaluated on a *current element*.

The axes *child* (which is the default axis if no axis is specified), *descendant*, and *descendant-or-self* (abbreviated as //) are called **forward** axes since they refer to the set of elements that are visited later than the current element in pre-order traversal of the document tree. Their counter parts *parent, ancestor*, and *ancestor-or-self* (abbreviated as **aos**) are called **reverse axes** since they refer to the set of elements that have been visited earlier than the current element in the pre-order traversal. We denote a^r as the counterpart of an axis a, e.g., $child^r$ is *parent*. Note that the axis *self* refers to the current element, and $self^r$ is still *self*.

2.2 Single-step XPath query

Every XPath query can be rewritten into an equivalent XPath query that contains only one location step and returns the same result set. Each subquery in the predicate can also be rewritten into a single step without changing its semantics. We call such an XPath query as a **single-step** query, in which the query itself and every subquery in it has only one location step. The rewritten process is based on the following two rules.

Subquery collapse For every subquery that has two or more location steps in the query in the form of $/a_1::n_1[p_1]/q$, we rewrite it into $/a_1::n_1[p_1$ `and` $q]$. If p_1 or q has two or more steps, this rule is applied again until every subqueries contains only one location step.

Step wrapping The following two XPath queries are equivalent:
$/A_1::N_1[P_1]/A_2::N_2[P_2]/Q$
$//N_2[P_2$ `and` $A_2^r::N_1[P_1$ `and` A_1^r`::ROOT]]`$/Q$
Since we can always append the implicit step `/self::ROOT` (`ROOT` nodetest matches the document root of every XML document), this rule can be applied to every XPath query. A single-step XPath query such as `/child::N` is rewritten into `//N[parent::ROOT]`. Therefore, for an XPath query (not subquery) in SSNF `/A::N[P]`, A is always //.

Note that `ancestor::ROOT (or aos::ROOT)` are true for every element in the document. Therefore, we can remove all `ancestor::ROOT and aos::ROOT` from the rewritten query and only keep `parent::ROOT` that are used in the query.

In an single-step query `/A::N[B]`, we call N as the unique **selecting nodetest** of it. B is in the form of $f(p_1, p_2, ...p_k)$ where p_i is a single-step query and f is a boolean function.

EXAMPLE 4. *Consider the following query:* `/Z//X[ancestor::Y[R]//S or //W[not(aos::T//K)]]` *First, the* subquery collapse *rule can be applied on the subquery* `ancestor::Y[R]//S`*, and the result if* `ancestor::Y[R and //S]`*. The* Step wrapping *can be applied to the query and the result is* `//X[P and aos::Z[parent::ROOT]`*, where* `P` *is the original predicate of* `X`*.*

2.3 Evaluation Tree

We model the XPath queries as a **evaluation tree** extended with a **node predicate** at each node. To distinguish the nodes in the evaluation tree and the nodes in the document tree, we refer the latter as elements in following discussion.

Unlike the syntax tree used by XPath evaluation algorithms, the evaluation tree separates the structure information and the logical information contained in the query. Therefore, for each element that may be matched in the query, we can determine easily which elements may affect its matching with the query, and which elements may be affected by the matching between this matching.

In a evaluation tree, each node is associated with a string as its label and a node predicate of all its children. We refer a node with label n as node n. (A simple rename process can solve conflicts caused by multiple nodes labeled with the same string.) P_n is used to denote the parent of node n. The edge between P_n and n is labeled with an axis a, denoted as A_n. The *node predicate* of node n, denoted as B_n, is a boolean function in which every child node of n is used exactly once.

To build the evaluation tree of query Q, we first transform Q into an equivalent single-step query Q'. We then transform Q' into a evaluation tree as follows. For a single-step query `/A::N[f(`p_1`,`p_2`,...`p_k`)]`, the root node of Q, denoted

as Q_r is labeled as N and the evaluation function B_{Q_r} is $f(n_1, n_2, ..., n_k)$ where n_i is the *selecting nodetest* in subquery p_i. In the sequel, we also call the subquery p_i as subquery n_i since the former is essentially the subtree rooted at n_i. Note that the nodetest n_i is also used as a variable name here. Assigning value TRUE to subquery n_i means that the variable n_i is set to TRUE in the evaluation of B_n.

An example of the evaluation tree of the example query used in Example 4 is depicted in Figure 1. The node-predicate are depicted as boolean expressions enclosed by the boxes. Nodes without explicit node predicate are either leaf nodes whose node predicates are empty or nodes with only one child `C` and the node predicates are $f(C) = C$.

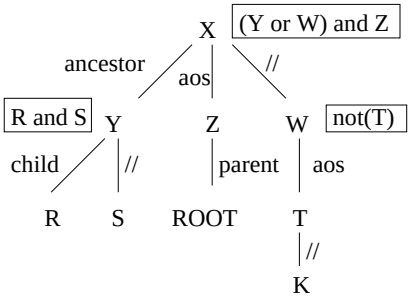

Figure 1: Sample query and evaluation tree

2.4 Evaluation of XPath Queries

The problem of XPath evaluation is defined as follows. Given an XML data tree T, an XPath evaluation tree Q, an element e in T **matches** (or *unconditionally matches*) a node n in Q if and only if e's name equals the label of n, *and* the node predicate of n B_n (if non-empty, otherwise always true) evaluates to TRUE given the following assignment: child node c of n (the edge between them is labeled as A_c) is assigned to TRUE if and only if an element e' can be reached via axis A_c from e and e' matches c; otherwise c is assigned to FALSE. We use $B_n(e)$ to denote the result of evaluating boolean function B_n given the assignment for every subquery in the predicate of e.

It is not hard to illustrate that the elements that match Q_r consist of the result set of the query. The process of the computing the matching is a bottom-up evaluation of the single-step query.

The above definition of matching is recursive. To determine the matching between an element e and an evaluation tree node n, we need to know for every child node c of n, whether there is an e' that matches c. In streaming XPath evaluation, we are limited to a single pre-order traversal of the document tree, and therefore such information is not always available.

In streaming XPath evaluation, an element e can **conditionally matches** an evaluation tree node n: if e has the same name as n, and $B_n(e)$ evaluates to NA (stands for pending results) with the following assignment for every child node c of n: c is assigned to TRUE if and only if an element e' can be reached via axis A_c from e and e' unconditionally matches c; c is assigned to FALSE if and only if there is not such element e'; if we cannot decide whether such element e' exists, c is assigned with NA, in which case we also say e is **pending** on subquery c. The evaluation of boolean expression with value NAfollows the three-valued logic.

There are two situations that c is assigned with value NA. Consider the time when we first encounter element e in the stream. For every child node c of n, if A_c is a forward axis, c should be set NA. Although there is not any descendant elements of e that matches c, it is possible that such element may come in the future. If A_c is a reverse axis, and c is *conditionally matched* by an ancestor of e, c should be assigned to NA as well.

Therefore, we distinguish the child nodes of n into two sets: the **forward set** that contains all the child nodes that connected to n via forward axes, and the **reverse set** that contains all the child nodes connected to n via reverse axes.

In the sequel, we explicitly distinguish the *unconditional matching* as in our original definition from the conditional matching. When we say element e matches node n, we mean only e's name is the same string as n's label.

3. OPTIMAL BUFFERING

In this section, we define what we mean by optimal buffering. The key idea is that a buffering strategy is optimal if, at any position in the stream, all elements in the buffer are elements that must appear in the buffer of all XPath query engines (that evaluate queries exactly). We refer to these elements as **useful elements**, and characterize them below.

To simplify the presentation, we will use a very simple model of the **buffer** in which each element is either entirely buffered (all information stored) or completely discarded (no information retained). Thus, the buffer may be thought of as a subset of the elements seen so far in the input. As discussed in later section, our method uses a more flexible model, permitting, for example setting bitmaps based on the presence of elements of a certain type. (In particular, our method buffers only a flag for each useful element that matches a non-root node of the query tree.) The definitions we present below can be easily adapted to that model and others like it.

We use $S(e)$ (and $E(e)$) to denote the positions of the at which the start (respectively, end) tag of an element e in the input stream. At any point in time, the prefix of the input data stream that has been encountered forms a **partial document tree**. We say an element is **open** if we have encountered its start tag but not its end tag. The rest of the elements in the partial document tree are **closed**. We maintain a stack of the open elements in the usual manner (push on start tag, pop on end tag). If T is a partial document tree of document tree T', we say T' is an **extension** of T (and T is a **prefix** of T') and use $X(T)$ to denote the set of all extensions of T.

We denote the result of evaluating an XPath query Q on tree T by $Q(T)$. For a partial document tree T and an XPath query Q, we use $D(Q, T)$ to denote the **definite result** of

Q on T. Intuitively, $D(Q,T)$ is the set of elements that are known to belong to the result based only the information in T, irrespective of the rest of the document tree. Similarly, we use $P(Q,T)$ to denote the **potential result** of Q on T. Intuitively, $P(Q,T)$ is the set of elements whose membership in $Q(T')$, for some extension T' of T, cannot be determined (positively or negatively) using only T.

DEFINITION 1. *For an XPath query Q and a partial document tree T, the **definite result** is defined as*

$$D(Q,T) = \{e \in T \mid \forall T' \in X(T) : e \in Q(T')\}$$

*The **potential result** is defined as*

$$P(Q,T) = \{e \in T \mid \exists T' \in X(T) : e \in Q(T') - D(Q,T)\}$$

We note $D(Q,T) \neq Q(T)$. For the query $Q =$ `A[not(B)]` and the tree $T = (R((A(B))(A)))$, $Q(T)$ consists of the second A element, but $D(Q,T)$ is empty because there are extensions of T in which that element is not in the result (e.g., $T' = (R((A(B))(A(B))))$).

We define a **null tag** $\perp$ that does not match any node-test in any XPath query. We use $T - e$ to denote the tree obtained by renaming e to $\perp$ in T. (In Section 4, we use the subroutine *setUseless(e)* to transform the current tree T to $T-e$.) The idea here is that if we replace all useless elements in a document tree by $\perp$, the evaluation of the query is *not affected*. To formalize what we mean by *not affected*, we define the **future results** for query Q, partial document tree T, and $T' \in X(T)$ as $F(Q,T') = Q(T') - D(Q,T)$. The useful nodes for evaluation tree Q at a time t when partial tree T has been encountered are defined as follows:

DEFINITION 2. *(**Useful and useless elements for partial document tree T and evaluation tree Q.**) The root of T is useful. Let $F(Q,T') = Q(T') - D(Q,T)$. An element $e \in T$ is useless iff $\forall T' \in X(T) : F(Q,T'-e) = F(Q,T')$; otherwise e is useful.*

From the definition, it follows that once an element e is deemed useless, it cannot become useful at some later point in the stream. The **expiration point** for an element e, denoted by $K(e)$, is the earliest point (in the stream) at which e is useless. A related concept is the **decision point** of an element. Suppose element e matches query-tree node n. The stream position, if any, at which e satisfies n's predicate is called the *positive decision point* of e. Similarly, the stream position, if any, at which e falsifies its predicate is called the *negative decision point* of e. If e does not match any evaluation tree node, its negative decision point is taken to be $B(e)$, its begin event.

DEFINITION 3. *The buffering performed by a streaming query evaluation method is **optimal** iff at each position in the stream the buffer contains only useful elements.*

We may detect an element is useless at various points in the stream. For some elements, we may detect that it is useless at time $S(e)$ (e.g., the element that does not match any node in the query tree). An element `Y` for query `//X/Y` is useless at point $S(Y)$ if we detect that the stack top is not an `X` element. For other elements, we may only detect that it is useless at a later time. For example, an element `X` for query `//X/Y` becomes useless only at $E(X)$ after which there can be no `Y` children of this `X` element can appear. As a slightly more complex example, consider the query `X[ancestor::Z[not(T)]` on the stream depicted in Figure 2. Consider the point in time just before element T_1 is encountered. The partial tree at this point is depicted in Figure 3. At point $S(T_1)$, all elements Z_1, X_2, X_3, and X_4 become useless because they cannot contribute to future result set any more.

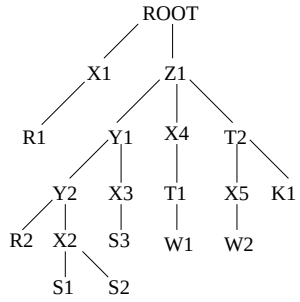

Figure 2: Sample data tree

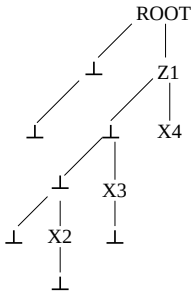

Figure 3: A modified partial tree

4. STREAMING XPATH EVALUATION

The evaluation algorithm is described as a pair of event handlers that respond to begin and end events in the stream. The pseudocode for these handlers is listed as Listings 1 and 2, with pseudocode for the propagation and pruning subroutines appearing as Listings 3 and 4. For ease of presentation only, we ignore the text contents of elements in the following description. The actions for such events are very similar to those described here. (Our implementation supports such events and predicates on text contents.)

During the evaluation, consider the arrival of an element e in the stream. If e's label does not match any node in the evaluation tree, then e is not useful for query evaluation in any form (neither as potential result data nor as data that satisfies predicates in the query). Thus, no actions need be taken for e. (However, we need to process e's descendants separately.) Such *useless* elements are immediately removed from the buffer. We use *setUseless(e)* to denote the operation of removing a useless element from the buffer. This subroutine is also called when a useful element becomes use-

less, as described later (e.g., line 8 of Listing 1).

With every *useful* element e, we associate two arrays: An **assignment array**, $e.assignment$, records which subqueries in e's predicate have been satisfied. Let n be the query-tree node that is matched with e. For each child c of n, if the subquery represented by c is satisfied for e, then $e.assignment[c]$ refers to the element matching c that yields a true evaluation of this subquery. (If multiple elements yield a true evaluation, only the one encountered earliest is referenced.) If the subquery represented by c is not satisfied, $e.assignment[c]$ holds the truth value false if the subquery is known to be false and NA otherwise (if the subquery cannot be determined to be true or false at this point in the stream). When we evaluate $B_n(e)$ (the node-predicate of n, for element e) the variable c in B_n is assigned true, false, or NA depending on whether $e.assignment[c]$ refers to an element, stores false, or stores NA, respectively. (Recall, from Section [**To do**], that the node-predicate associated with a node of the evaluation tree is different from the predicate of the corresponding location step of the query. For example, for the location step `X[Y//Z and //R/S]]`, B_X is `[Y and //R]`.) The second array associated with each element e is the **pending array**, $e.pending$. This array indicates, for each pending subquery c of e, the elements (possibly none) that can be used to satisfy c. For all c, $e.pending[c]$ is initially the empty set and is nonempty iff $e.assignment[c]$ is NA.

In the event handler **BeginHandler** (Listing 1), we process an element e whose label matches the label of evaluation tree node n. Element e is useful only when n's node-predicate for e, $B_n(e)$, evaluates to true or NA and e can be used by some useful elements (either current or future). We evaluate $B_n(e)$ as follows. All subqueries in B_n that test e's descendants are set to NA. The value of each subquery c in B_n that tests e's ancestors is determined using the **checkStack(c)** subroutine. This subroutine first checks the stack for an ancestor e' of e that matches c (unconditionally). If such an e' is found, checkStack(c) returns true. Otherwise, the subroutine checks whether there is an ancestor e' that conditionally matches c. If such an e' is found, it returns NA. Otherwise, it returns false to denote that no ancestor satisfies the subquery c for element e. A similar subroutine **pendingInStack(c)** returns elements from the stack (which are all ancestors of the current element e) that conditionally match c.

If $B_n(e)$ evaluates to false, we can ignore element e since it falsifies its predicate. If the result is either true or NA, e can be useful if one of the following conditions holds: (1) Node n's axis A_n (connecting n to its parent P_n in the evaluation tree) is a forward axis and some ancestor (in the document tree) e' of e requires e in its predicate. In more detail, we use the **requiredInStack(n)** subroutine to scan the stack for an ancestor that may use e to evaluate its subquery, n. For each element e' in the stack whose label matches n's label, e is added to e''s pending array iff $e'.assignment(e)$ is NA (implying e''s subquery n is unsatisfied). (2) Axis A_n is a reverse axis. In this case, e may be used by its descendants, which have not yet been encountered because of the preorder serialization of the document tree.

The event handler **EndHandler** is mainly used to address the existential quantification semantics of XPath predicates. At the end event of element e, every e's pending subquery c that tests e's descendants will be set to FALSE(unless some pending descendants conditionally matches c and is waiting for result from e's ancestors).

Consider an element e that unconditionally matches a evaluation tree node n. If n is Q_r, the root node of the evaluation tree, e is a definite result item. We can simply set e as useless (as illustrated in line 5 in Listing 2) since all we cannot generated new result content from e and e will not be used in the predicate of future elements. If n is not Q_r, e must be used in the evaluation of the predicate of other element that matches P_n, the parent node of n. If the axis connects n to its parent P_n (A_n) is a forward axis, e must be used in the predicate of its ancestor elements. At least one of the ancestors is a useful element, otherwise e cannot be useful. Since e is still used by its ancestor, we do not need to perform any action for this event. If A_n is a reverse axes, e is used in its descendants. Unless there are descendants of e are still useful, e can be deemed as useless, as in line 9 in Listing 2) .

At the end of a pending element e, every pending subquery that tests a descendant of e should be falsified. However, in the presence of reverse axes, it is possible that a descendant of e is waiting for the result from an ancestor of e that is still undecided at the end of e. This event handler deals with both cases and re-evaluates predicates for e as needed.

In Listing 2, from line 13 to line 17, we set to FALSE every subquery that is still unsatisfied *and* not depend on any descendant elements (and therefore cannot be satisfied in the future). The predicate of e is evaluated if there are subquery changed form NA to FALSE in the previous step. If the result is: (1) TRUE the processing is very similar to the case when we encounter an element that unconditionally matches a node, except that the *new* result needs to be propagated; (2) FALSE, e has been proved to be useless; (3) NA, e is still useful since e is useful before this event and its predicate result is unchanged. These cases are handled in Listing 2 from line 18 to line 36.

When an element e reaches its positive decision point, we emit it as a result item if e matches Q_r, the root of the evaluation tree. If e matches a non-root interior node n then it either (1) is currently used by some other element whose subquery n is pending (when A_n is a forward axis) or (2) will be used in the future (when A_n is a reverse axis). (Otherwise e would not be useful.) In the first case, e satisfies the subquery n for those elements, which may change the predicate result for those elements. The change may make those elements reach their decision point and perhaps become useless. The **propagation** subroutine (Listing 3) outlines how usefulness information is updated in this manner.

Using the arrays *assignment* and *pending*, we can also determine the used-by relationships between useful elements. Only open useful elements (elements whose end tags have not been encountered) can be used directly to satisfy a subquery for any future element. Closed useful elements are useful only because they are used by other useful (buffered) elements. Therefore, when a useful element e becomes use-

less, all the useful elements used *only* by e also become useless. The **pruning** subroutine (Listing 4) outlines how uselessness information is updated in this manner.

5. PROOF OUTLINE

Detecting new useful elements Once an element is deemed useless, it remains useless. Therefore, new useful elements appear only at begin events. Consider the arrival of the begin tag $B(e)$ of element e assuming that only useful elements are currently in the buffer. If e does not match any node in the evaluation tree then e does not affect the query evaluation in any way and is simply ignored with no changes to the buffer. If e does match an evaluation tree node n, e is useful in three cases: (1) e is a result item or a potential result item; (2) e is used in the predicate of some current pending useful elements to satisfy their pending subquery n; (3) e may be used in the predicate of future useful elements. All three cases are handled by the procedure in Listing 1. For the first case, it is only possible if n is the root of the evaluation tree. This case is handled from line 3 to line 15. For the second case, only the ancestors of e (since e's descendant cannot come before e) can use e in their predicate if they conditionally match evaluation tree node P_n, which is the parent node of n. Since e is the new element, all its ancestors should be open and therefore in the stack. As illustrated in line 23, we deem this new element as useless if no element in the stack requires it. For the third case, since e is used in the predicate of future element, e must be used an ancestor in the predicate (since we do not allow sibling or following axis). This case is handled in 16. In all cases, we need to evaluate the predicate for e. Only when the evaluation result is TRUE or NA can we deem the new element as useful. As we described before, an element that falsifies its predicate will not be used in the evaluation because of the existential semantics of XPath.

Detecting expiration points After we detect the useful element at the time we encounter them, we need to illustrate that we properly detect their expiration point, i.e., the point a useful element becomes useless.

A useful element e that unconditionally matches a node n (i.e., e has already reached its positive decision point) may reach its expiration point in the following cases: (1) e is a result item and e is closed; (2) e is used in another useful element e's predicate and e' is no longer useful; (3) e may be used in future element's predicate, but e is closed and not used by any current useful element.

The first case is simple and handled in line 5 in Listing 2. The second case can be only caused by propagated results. It will be discussed in more detail later in this section. We now only discuss the expiring elements not caused by the propagated results. The third case is handled in line 9 in Listing 2. In this case, no future element will use e in their predicate since e may only be used in its descendant elements. We do not need to prune the other current useful elements because none of them is using e in their predicate evaluation (i.e., in their assignment array.) Otherwise e will not be deemed as useless.

A useful element e that conditionally matches a node n (i.e., e has not yet reached its decision point) also reach its ex-

Listing 1 Handler for begin events

```
1: {Process the begin event of element e whose label matches
   that of query-tree node n.}
2: BeginHandler(Element e, Node n)
3: if n = Q_r then {n is query-tree root}
4:   getInitAssignment(e, n);
5:   if B_n(e) = TRUE then
6:     emit e; {e is a result item}
7:   else if B_n(e) = FALSE then
8:     setUseless(e); {ignore e henceforth}
9:   else {B_n(e) = NA}
10:    for every c in n.ReverseSet do
11:      if checkStack(c) = NA then
12:        e.pending[c] ← PendingInStack(c);
13:      end if
14:    end for
15:  end if
16: else if A_n is a reverse axis then {A_n is n's axis}
17:   {e is useful for future elements}
18:   getInitAssignment(e, n);
19:   if B_n(e) =FALSE then
20:     setUseless(e);
21:   end if
22: else {A_n is a forward axis}
23:   if not requiredInStack(n) then
24:     setUseless(e);
25:   else
26:     getInitAssignment(e, n);
27:     if B_n(e) = FALSE then
28:       setUseless(e);
29:     else if B_n(e) = TRUE then
30:       for every e' in stack pending on n do
31:         checkPending( e, e', n );
32:         e'.assignment[n] ← e;
33:       end for
34:       propagation(e, n);
35:     else {B_n(e) = NA}
36:       for every e' in stack pending on n do
37:         e'.pending[n] = e'.pending[n] ∪ e;
38:       end for
39:     end if
40:   end if
41: end if
42:
43: {Scan the stack for e's ancestors that are used in n's predi-
   cate.}
44: getInitAssignment(Element e, Node n)
45: for every c in n.ReverseSet do
46:   e.assignment[c] ← checkStack(c);
47: end for
48: for every c in n.ForwardSet do
49:   e.assignment[c] ← NA;
50: end for
51:
52: {When a subquery n of e' is satisfied by e, the previous ele-
   ments in e'.pending[n] may become useless.}
53: checkPending( Element e, Element e', Node n)
54: for ∀e'' ∈ e'.pending[n], e'' ≠ e do
55:   if ∄x : x ≠ e' ∧ e'' ∈ x.pending[n] then
56:     setUseless (e");
57:     pruning( e", n );
58:   end if
59: end for
```

Listing 2 Handler for end events

```
1: {Process the end event of an element e that matches query-tree node n.}
2: EndHandler(Element e, Node n)
3: if B_n(e) = TRUE then
4:   if n = Q_r then {root of the evaluation tree.}
5:     setUseless(e);
6:     pruning(e, n);
7:   else if A_n = reverse axis then
8:     if ∄e' : e'.assignment[n] = e then
9:       setUseless (n); {no pruning}
10:    end if
11:  end if
12: else if B_n(e) = NA then
13:   for every c in n.ForwardSet do
14:     if e.assignment[c] = NA ∧ e.pending[c] = ∅ then
15:       e.assignment[c] ← FALSE
16:     end if
17:   end for
18:   if e.assignment is changed then
19:     if B_n(e) = FALSE then
20:       setUseless (e);
21:       pruning(e , n);
22:     else if B_n(e) = TRUE then
23:       if n =Q_r then
24:         emit e;
25:         setUseless(e);
26:         pruning(e, n);
27:       else {n is used in other predicates.}
28:         propagation(e, n);
29:         if A_n = reverse axis then
30:           if ∄e' : e'.assignment[n] = e then
31:             setUseless (e);
32:           end if
33:         end if
34:       end if
35:     end if
36:   end if
37: end if
```

Listing 3 Propagation of usefulness information

```
1: {n is not the query-tree root and B_n(e) = TRUE.}
2: Propagation(Element e, Node n)
3: for every e' such that e ∈ e'.pending[n] do
4:   {e' matches n's parent P_n in evaluation tree}
5:   checkPending(e, e', n);{subquery n for e' is satisfied by e}
6:   e'.assignment[n] ← e; {eval. for e' uses e}
7:   if B_{P_n}(e') = TRUE then
8:     if P_n = Q_r then
9:       emit e' as a result item;
10:      setUseless(e');
11:      pruning(e', P_n); {propagate uselessness from e'}
12:    else
13:      propagation(e', P_n); {recurse up evaluation tree}
14:    end if
15:  else if B_{P_n}(e') = FALSE then
16:    setUseless(e');
17:    pruning(e', P_n);
18:  end if{skip case B_{P_n}(e') = NA}
19: end for
```

Listing 4 Pruning useless elements

```
1: {Element e is falsified or deemed useless and n is a non-leaf query-tree node}
2: Pruning(Element e, Node n)
3: for every c ∈ n.ForwardSet ∪ n.ReverseSet do
4:   x ← e.assignment[c];
5:   {e uses x}
6:   if ((∀e' matching n : (e' ≠ e → x ≠ e'.assignment[c])) ∧
     (c ∈ n.ForwardSet ∨ x is closed)) then
7:     setUseless (x); {x is not used for anything else}
8:     pruning(x, c); {recursive pruning of subtree}
9:   end if
10: end for
11: for every x that matches P_n do
12:   x.pending[n] ← x.pending[n] − {e};
13:   if x.pending[n] = ∅ and A_n = reverse axis then
14:     x.assignment[n] ← FALSE;
15:   end if
16:   if B_{P_n}(x)=FALSE then
17:     x . setUseless(x);
18:     pruning(x, P_n);
19:   else if B_{P_n}(x) = TRUE then
20:     propagation(x, P_n);
21:   end if
22: end for
```

piration point in the above three cases. The difference in the processing of case (1) and case (3) is that we need to re-evaluate the predicate for e. Since e is closed, every pending subquery of e that tests the descendant of e is now falsified.

For the case (1), if the evaluation result is TRUE, e is now a definite result item and is returned to the user. It is also deemed as useless afterward. If the result is NA, e must be still pending on some ancestors since we have assigned all pending subqueries testing descendants to FALSE. Therefore, e is still possible to be a result item, and thus till useful. If the result is FALSE, e reaches its negative decision point and is deemed as useless.

In case (3), e is always useless after the evaluation since it is not needed after the evaluation, and it can not be used in the predicates of future elements. The evaluation is needed if there are elements pending on e, i.e., have e in their pending array.

Moreover, a useful element e that conditionally matches a node n may become useless if e reaches its negative decision point. Since we do not consider the propagated result, we consider every element that reaches its negative point in its own begin event or end event. For elements that are falsified in its begin event, the case is simple since no other useful element will be affected by this new element, as in line 19 and line 27in Listing 1. (Recall that an element is useful for all possible extension of the current partial document tree.) For element that is falsified in the end event because of the new assignment for pending subqueries testing descendants, we also need to prune other elements that are falsified by this fact, as illustrated in line 19 in Listing 2.

Decision Propagation Consider a useful element e that matches a node n and $B_n(e)$ evaluates to NA reaches its decision point. The decision point could be either the positive decision point when $B_n(e)$ evaluates to true, or negative decision point when $B_n(e)$ evaluates to false. In either case

some current useful elements may become useless. Here we only locate those elements directly affected by e's decision point. Other affected elements can be located recursively in similar manner.

The *pruning* process is performed when an element e conditionally matches evaluation tree node n reaches its negative decision point. An other useful element e' may also reach its decision point (either negative or positive) because of this fact. First, e cannot be matching a leaf-node n: since n has no predicates, e's decision point is always at time $B(e)$ (and therefore cannot conditionally match n at first and reach the decision point later).

When e matches a non-leaf node n reaches its negative point, first we consider n's child nodes in the evaluation tree. Since e is useful before the current point, $B_n(e)$ evaluates either to TRUE or NA.

The first set of useful elements that may be affected by the falsifying e are elements that match n's child node and therefore are used in the evaluation of $B_n(e)$. If an element in this set is e's descendant, it becomes useless as well since it is used only in e's predicate and can no longer affect other elements. This case is handled from line 3 to line 10 in Listing 4. Note that it is possible that some descendant elements of e used in the evaluation of $B_n(e)$ are used by some ancestor e' of e that matches the same node n. These descendant elements will stay useful. If an element in this set is e's ancestor, they are still useful if either they are open in which case they may be used by future siblings of e that also match n, or they are closed but are used in the evaluation of $B_n(e')$, where e' is a useful element that matches n as well. This case is handled from line 3 to line 10 in Listing 4.

The second set of useful elements that may affected by the falsifying e are the elements that match n's parent in the evaluation tree and thus use e in their predicate evaluation. This case is handled from line 11 to line 22 in Listing 4.

Since e is useful before this point, there must exists an element e' that matches P_n must be waiting for the result of $B_n(e)$ (otherwise e is useless). e' may also waiting for other useful elements that also matches n. If there exists such e'' that matches n and is depended by e', $B_n(e'')$ should also be NA, otherwise e becomes useless at the point when $B_n(e'')$ evaluates to true (processed in the Listing 3 at line 5). In this case, e' is still useful and waiting for the result of $B_n(e'')$. If there does not exist such e'', i.e., e' is only waiting for e for the result of subquery n. We have to examine the axis A_n that connects n and P_n. If A_n is a reverse axis, e' cannot have future ancestor that matches n, therefore subquery n is set to false for e' and $b_{P_n}(e')$ is reevaluated. If A_n is a forward axis and e' is still open, e' may have future descendant that matches n. Therefore, subquery n for e' is still NA and e' is still useful (since $b_{P_n}(e')$ is not re-evaluated). If A_n is a forward axis and e' is closed, e' can not have future descendant that matches n. Therefore, subquery n is set to false for e' and $b_{P_n}(e')$ is reevaluated.

The above *prune process* is applied recursively to all elements that are once useful and now reach their negative decision point. It termination is guaranteed by the simple fact we visit every element at most k times, each time when one of its subqueries is satisfied.

Consider a useful element e that matches a node n and $B_n(e)$ evaluates to NA. When it reaches its positive decision point, some useful element may become useless as well.

There are two sets of element that are affected by e's positive decision point. The first set is elements that use e in their predicate evaluation. The second set is elements that are used by e in e's predicate evaluation.

For an element e' in the first set, e must in e'.pending[n] since e is previously pending. We can determine two facts here: (1) Every other element in e'.pending[n] is no longer required for e' to evaluate its predicate. Therefore, they should become useless if they are not used by other elements. This fact is handled from line 54 to 59 in Listing 3. (2) The predicate of e' needs to be re-evaluated since one of its subquery is newly satisfied. Based on the new evaluation result, e' should be processed accordingly and the result may need to be further propagated. This fact is handled from line 6 to line 18 in Listing 3.

Consider an element e' in the second set that matches a child node c of node n. e' must be in e.assignment[c]. e' will be useful if e is still useful. e' will become useless when e becomes useless and e' is not used by other useful elements, which is handled by the *pruning* process described earlier.

Note that the propagation and pruning process may be interleaved. As we show during the reasoning, a falsifying element may evaluate the predicate of some pending elements to true, and a satisfying element may evaluate the predicate of some pending elements to false. In any case, we can see that the useful elements are correctly identified.

6. IMPLEMENTATION

Matching Stack An important operation in our method is to scan the stack to look for ancestors with certain features. For example, in line 46 in Listing 1, we have to check the stack for every child node c of n, whether there is an ancestor in the stack matches c (conditionally or unconditionally). Such checking requires $O(d)$ time in the stack, where d is the depth of the stack. In the implementation, we store a **matching stack** for every node c in the evaluation tree. In the matching stack of node c , we store the pointers to the elements in the global stack that match c (both conditionally and unconditionally) and keep a flag to denote whether any of them unconditionally matches c. As a result, the stack checking operation requires only constant time. The space requirement is $O(d+q)$ where q is the size of the query.

Dependency Graph For every useful element e that matches a evaluation tree node n, we have used explicit arrays *e.assignment* and *e.pending* and an implicit **dependent set** $e.dependents = \{e' | e \in e'.pending[n]\}$, which indicates the elements that depend on e. Maintaining the dependent set incurs a very small overhead because whenever we add e to e''s pending array, we add e' to e's dependent set. The two arrays and the implied dependent set encode all the dependencies among the useful elements. These dependen-

cies form a directed acyclic graph, called the **dependency graph**, with a node for each useful element and edges denoting dependencies. There is an edge from element e to element e' iff e is in the assignment array or the pending array of e', i.e., e is used or may be used in e''s predicate evaluation. In other words, the evaluation tree node n that e matches is the child of the evaluation tree node n' that e' matches, i.e., $n' = P_n$. Moreover, whether e' matches P_n is depending on whether e matches n. The matching between e and n may be conditional, in which case $B_{p_n}(e')$ must evaluate to NA, and $e \in e'.\text{pending}[n]$; or unconditional, in which case $e = e'.\text{assignment}[n]$.

7. RELATED WORK

XPath evaluation has been the focus of much recent attention. Gottlob, Koch, and Pichler recently presented a polynomial time algorithm for evaluating XPath [11]. Their algorithm, based on dynamic programming, provides an $O(D^5Q^2)$ time bound and $O(D^4Q^2)$ space bound (where D and Q denote data and query sizes, respectively). They also provide a linear-time algorithm for an XPath fragment they call *core XPath*, which includes structural and logical constraints on the document tree but does not include value comparisons. These algorithms use the fact that an XPath query can be rewritten in $O(Q)$ time into an equivalent, simpler XPath query that uses only two types of axes: *first-child* and *next-sibling*. These two axes, including their reverse axes and their combinations, can be evaluated over any document tree in $O(D)$ time. However, these methods are difficult to extend to streams because the necessary tuples of the *next-sibling* relation are, in general, not available in a stream when needed. In follow up work [9], these authors improved the time bound to $O(D^4Q^2)$ and space bound to $O(D^2Q^2)$. They also provide a method to evaluate a larger and very practical fragment of XPath that can process arithmetic and position functions (called Extended Wadler Fragment) in $O(D^2Q^2)$ time and $O(DQ^2)$ space.

There has also been work exploring the complexity of different subsets of XPath [12]. Core XPath, although it can be evaluated in linear time, is P-hard and therefore difficult to parallelize. Further, without negation, an even larger fragment, with arithmetic and position functions, can be evaluated with LOGCFL combined complexity. Segoufin has studied the complexity of typing XML documents and querying XML documents using core XPath with both DOM- and SAX-like encodings [29]. The data complexity is proved to be uTC^0 for a SAX-like encoding, and LOGSPACE for a DOM-like encoding. The paper also proves that the combined complexity for core XPath without negation is LOGCFL.

Neven and Schwentick have demonstrated the use of monadic second-order logic (MSO) for specifying complex patterns over trees [25]. Gottlob and Koch have pointed out that MSO is able to capture the node-selecting function of XPath and can be used to investigate properties of XPath queries [8]. The paper shows the analogies between XPath and MSO queries and illustrate how to transform MSO queries into simple acyclic rules.

Neven and Schwentick propose the use of a **query automaton (QA)**, which is a tree automaton extended with a selecting function, to evaluate queries over ranked and unranked trees [24]. The query automaton can capture all unary queries expressible using MSO, and require a special stay transition for unranked trees. In a query automaton, the state of a parent may be changed only when the state of all children is known, and states for all the children are assigned at the same time when the down transition takes place. For streaming data, these two transitions may not be applicable directly. We may specify a dummy child that corresponds to all unknown children, and thus allow both transitions in the incomplete document tree.

Frick, Grohe, and Koch have proposed a bottom-up tree automaton with a selection function, called **selecting tree automaton (STA)** to evaluate XPath queries over binary trees that encode the original trees using the standard first-child/next-sibling method [20]. Koch has also provided a two-pass evaluation algorithm for XPath [19]. This algorithm first translates an XML document into a binary tree and the XPath query into a tree-marking normal form program. The program is then evaluated over the binary tree using an STA in two passes: In a bottom-up pass, all possible reachable accepting states for every node are computed. In the top-down pass, the conditions specified by the program are computed for every node in order to prune the states. Such a two-pass algorithm is a useful relaxation of the single-pass methods we study, and merits further study.

Another automaton that has been studied in an XML context is the tree-walking transducer. Although it is has not been used to evaluate XPath query (to the best of our knowledge), it has been applied to model programs in XML transformation languages such as XSLT. Milo, Suciu, and Vianu have defined a k-pebble tree-walking tree-transducer model for XML transformation that can be used to check whether the result of a transformation conforms to a DTD [22]. Neven has proved that tree-walking automaton to be not relational complete [23] even with relational storage (rather than simple registers) and look-ahead operation. However, such automaton is also proved to be able to catch all unary MSO queries, and thus may be used in XPath evaluation. Although streaming XML is traversed in depth-first order, some transition combinations such as visiting previous siblings, are not always applicable. We speculate that the tree-walking machine, given some extensions, may be a good tool for evaluating more complex XPath queries over streams.

Several systems have developed for streaming XPath processing. Broadly, they fall into two categories: filtering and querying. In a filtering system, often used in a publisher-subscriber scenario, an XML document (in its entirety) in the stream is returned to a user if the document matches the XPath query specified by the user. Thus, no buffering is needed to store potential result items. (More precisely, such systems either return document identifiers, in which case no buffering is needed, or documents, in which case at most one document needs to be buffered at any time.) The key challenge in a filtering system is evaluating a very large number of queries simultaneously. The XFilter system [1] translates XPath queries (without predicates) into finite-state automata that are are indexed to permit efficient filtering for all queries simultaneously. The YFilter system [6] extends Xfilter's method to use common prefixes of XPath queries to combine the automata into a single automaton.

It also supports conjunctive subqueries in predicates. The XPush machine [14] uses an alternating automaton to evaluate XPath queries with predicates. By training the machine using sample documents, the throughput of the system can scale up very well to support a very large number of queries. Segoufin and Vianu have proposed mapping DTDs to pushdown automata for validating streaming XML [30]. This problem can be considered as a filtering problem as well, since the pushdown automaton can filter out documents that satisfy the DTD.

Several XPath querying systems have also been developed. XMLTK [13] is a set of XML tools developed at University of Washington. The xrun program in the toolkit can evaluate XPath queries on large XML datasets. It uses a DFA generated from the XPath query that takes the SAX events as input and return the results of the query. The **XAOS** system [3] can evaluate XPath queries with reverse axes and predicates without negations over streaming data. It uses two data structures called *X-dag* and *X-tree* to filter out the *related* elements that may be used in the evaluation of the query. At the end of a document, the query is evaluated by traversing the two data structures. Our method, unlike XAOS, focuses on the optimal buffering and optimal predicate evaluation. On the other hand, the XAOS method is simpler.

The streaming XQuery query engine described in [7] uses an iterator-based approach in which each function and operator is implemented as an iterator. An iterator consumes the output streams from its input iterators and produces a single stream, which may used as the input of the other stream. XPath expressions are also implemented in the form of XPath steps using iterators. It would be interesting to investigate whether we can optimize such iterator-based techniques for the much simpler XPath queries and compare the performance with other methods (such as our method and the other automaton-based methods).

The evaluation of XPath queries is closely related to the problem of tree pattern matching. Miklau and Suciu point out [21] that XPath evaluation is essentially a different problem than the *classical tree pattern matching* [15] and the *unordered tree inclusion* [18] problems. Most algorithms provided for the latter problems require post-order traversal of the data tree. Hoffmann and O'Donnell provide a top-down algorithm for the classical tree pattern matching problem [15]. The algorithm needs only a preorder traversal of the data tree. However, since it allows only parent-child edges in the pattern and preserves the order of siblings in the pattern, the algorithm cannot be directly applied to the patterns that contain ancestor-descendant edges and do not imply orders between siblings in the patterns (as is the case with XPath).

8. PERFORMANCE EVALUATION

The goal of performance study is to examine the throughput, memory footprint, and output latency of our method. We also compare the performance of XSQ with other systems that can process XPath queries: **Saxon** (`http://saxon.sourceforge.net/`), **Xalan** (`http://xml.apache.org/xalan-j`), XPATH from XMLTaskForce (`http://www.xmltaskforce.com`) (**XXTF**), and **XMLTK** [2]. XMLTK (version 1.01) does not support XPath queries that need buffering. Therefore, we use it only in experiments using queries without predicates and reverse axes. Saxon (version 6.5.2) and Xalan (version 2.4.0) are two broadly used high performance XSLT processor. XXTF is the implementation from XMLTaskForce of the polynomial algorithms introduced in [10]. All three systems are main memory systems that need to build the DOM tree in the main memory before evaluation.

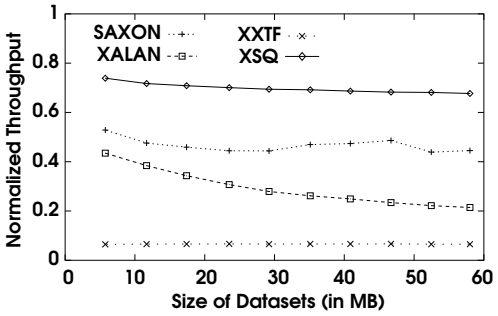

Figure 4: Throughput for XMark datasets and a query without reverse axes: `//regions/samerica[//payment and //mailbox[//from]]//item[quantity>=2 or shipping]/name`

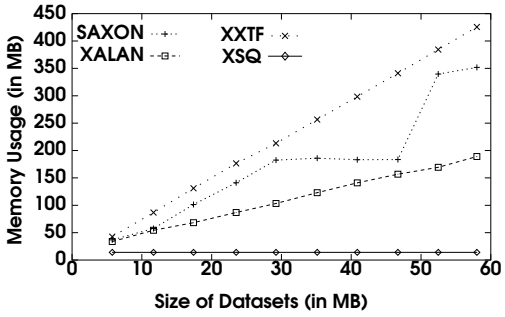

Figure 5: Memory footprints for the results of Figure 4

Metrics For each system tested in the experiments, we measured the normalized throughput and maximum memory usage.

The normalized throughput of the systems is the raw throughput of the system normalized by the throughput of the corresponding *pure parser* that parses the data but does nothing else. We wrote a pure parser in Java using Xerces2 Java Parser 2.4.0 Release, which is used as the XML parser for XSQ, Saxon, and Xalan in the experiments. We also wrote another pure parser in C using Expat parser, which is used by the XPATH program. XMLTK 1.01 provides a parser program named XParse that parses the document and count the number of elements in the document. Since the counting process only needs one statement at the begin event of every element, which should be very fast operation compared to the parsing process, we use the XParse program as the pure parser of the XMLTK system.

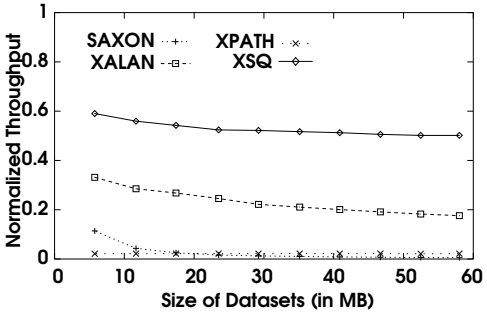

Figure 6: Throughput for XMark datasets and a query with two reverse axes:
`//listitem[ancestor::item[descendant::price>10 or descendant::quantity=1] or ancestor::annotation[happiness>8]]`

The maximum memory usage is the largest size of the memory that is allocated by the system during the evaluation. For the Java-based systems, the memory allocated by JVM is included in this value.

Setup We generated ten datasets using XMark benchmark program with the scale factor set to 0.5 to 5, step by 0.5. The sizes of result dataset range from 5.7MB to 58MB. We ran all the experiments in a Pentium III 900MHz PC with 1 GB of main memory running Redhat 7.2 distribution of GNU/Linux (kernel 2.4.9-34). The maximum memory the Java virtual machine can use was set to 512MB. The running time is obtained using the GNU TIME(1) tool. Every data point presented is obtained from the average of the results of ten runs.

Varying sizes of the dataset We first tested the systems on various sizes of data. Figure 4 depicts the normalized throughputs of the systems evaluating an XPath query without reverse axes. Since XSQ's running time is linear to the size of the data, its throughput is usually constant for different size of data.

Figure 5 depicts the memory usages of the systems in the same set of experiments. The other three systems are main memory systems that need to build the DOM tree before evaluation. The linear memory usage of them is as expected. (It is not clear why the memory usage for Saxon has the curve as depicited in the figure.) For XSQ, only in the worst case, where a predicate can be evaluated at the end of the document, it is required to buffer all the data.

Figure 6 illustrates the normalized throughput of the systems evaluating a complex query that contains two reverse axes in the predicates. We can see that the performance of Saxon and XXTF degrades because of the reverse axes, which implies more intermediate result for XXTF and more document tree traversals for Saxon. XSQ and Xalan, however, are not affected as much. The figure also illustrates that, even for complex queries, the running time of XSQ is still linear to the size of the data.

Varying size of the query As we described in Section 6,

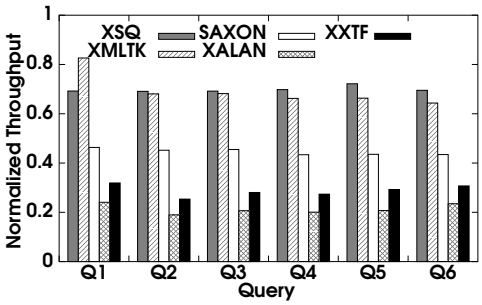

Figure 7: Simple Query of Various Size.
Q1: /site/regions/samerica/item/name
Q2: //site//regions//samerica//item//name
Q3: //regions//samerica//item//name
Q4: //regions//item//name
Q5: //regions//name
Q6: //name

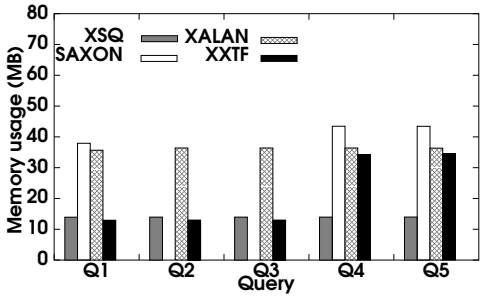

Figure 8: Memory usage for query with reverse axes in the main trunk.
Q1: //text
Q2: //listitem/a::parlist/a::site//text
Q3: //listitem/a::parlist/a::description/a::site//text
Q4: //listitem/a::parlist/a::description/a::item/a::site//text
Q5: //listitem/a::parlist/a::description/a::item/a::regions/a::site//text
Note: (1)Dataset size : 5.7MB.
(2)a:: is abbreviation for ancestor::.
(3)Saxon ran out of memory for Q2 and Q3.
(4)For Q4 and Q5, the scale for Saxon is 1/10.

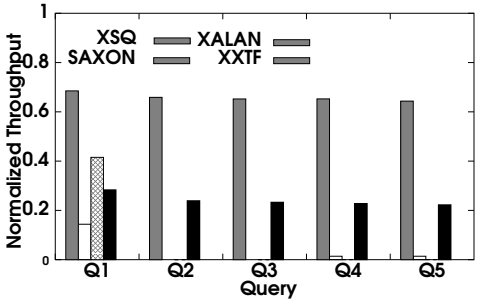

Figure 9: Normalized throughput for queries in Figure 9.

the throughput of XSQ is not affected by the total size of the query and the closure axes of the query. Figure 7 illustrates the throughput of the systems when evaluating queries without predicates but with different number of location steps. Recall that XSQ always rewrite the query into a single step query (e.g., `//regions//name` → `//name[aos::regions]`), which seems to complicate the query. However, as Figure 7 illutrates, XSQ's throughput does not change much when the query size increases (which introduces more reverse axes). We also note that XMLTK performs best for queries without closure axes, for which it can use a deterministic automaton.

Varying Number of Reverse Axes We also tested the systems for various number of reverse axes in the query. Figure 8 illustrates the memory usage for different systems when evaluating a set of queries that contain different number of reverse axes in the main trunk of the query. Figure 9 depicts the throughput of the systems for the set of queries. XSQ's running time and memory usage are also not affected by the number of reverse axes, as illustrated in this two figures.

For XXTF, since it evaluate the query step-by-step, larger intermediate result set is created for some queries. A closer look at the dataset shows that Q4 and Q5 generates significantly larger intermediate result set because of the large number of `item` element in the dataset which are selected by the two queries.

For the XSLT engines, they need traverse the document tree multiple times to evaluate the reverse axes. Therefore, more reverse axes imply more traversals, which lead to longer evaluation time and generates more intermediate result. However, we also note that Xalan seems to treat reverse axes in the main trunk and in the predicate differently. For the latter, the performance of Xalan is much better (cf. Figure 6). The experiments on different number of reverese axes in the predicate, is not presented here due to the space limit.

9. REFERENCES

[1] M. Altinel and M. J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 53–64, September 2000.

[2] I. Avila-Campillo, D. Raven, T. Green, A. Gupta, Y. Kadiyska, M. Onizuka, and D. Suciu. An XML Toolkit for Light-weight XML Stream Processing, 2002. `http://www.cs.washington.edu/homes/suciu/XMLTK/`.

[3] C. M. Barton, P. G. Charles, D. Goyal, M. Raghavachari, V. Josifovski, and M. F. Fontoura. Streaming XPath Processing with Forward and Backward Axes. In *Proceedings of the International Conference on Data Engineering*, March 2003.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language 1.0. W3C Working Draft, W3C, `http://www.w3.org/TR/xquery/`, August 2003.

[5] J. Clark and S. DeRose. XML path language (XPath) version 1.0. W3C Recommedation, W3C, `http://www.w3.org/TR/xpath`, Nov 1999.

[6] Y. Diao, P. Fischer, and M. J. Franklin. YFilter: Efficient and Scalable Filtering of XML Documents. In *Proceedings of the International Conference on Data Engineering*, pages 341–344, February 2002.

[7] D. Florescu, C. Hillary, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, August 2003.

[8] C. K. Georg Gottlob. Monadic Datalog and the Expressive Power of Languages for Web Information Extraction. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 17–28, Madison, Wisconsin, June 2002.

[9] Georg Gottlob, Christoph Koch, and Reinhard Pichler. XPath Query Evaluation: Improving Time and Space Efficiency. In *Proceedings of the International Conference on Data Engineering*, March 2003.

[10] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, Aug. 2002.

[11] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Hong Kong, China, August 2002.

[12] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 179–190, San Diego, California, June 2003.

[13] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with Deterministic Automata. In *The 9th International Conference on Database Theory*, pages 173–189, Siena, Italy, January 2003.

[14] A. K. Gupta and D. Suciu. Streaming processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 419–430, San Diego, California, June 2003.

[15] C. M. Hoffmann and M. J. O'Donnell. Pattern matching in trees. *Journal of the ACM (JACM)*, 29(1):68–95, 1982.

[16] IPEDO. `http://www.ipedo.com`, 2002.

[17] J. X. Josephine M. Cheng. Xml and db2. In *Proceedings of the International Conference on Data Engineering*, pages 569–573, San Diego, California, February 2000.

[18] P. Kilpel. *Tree Matching Problems with Applications to Structured Text Databases*. PhD thesis, Dept. of Computer Science, University of Helsink, 1992.

[19] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, September 2003.

[20] Markus Frick and Martin Grohe and Christoph Koch. Query Evaluation on Compressed Trees. In *LOGIC IN COMPUTER SCIENCE*, Ottawa, Canada, June 2003.

[21] G. Miklau and D. Suciu. Containment and Equivalence for an XPath Fragment. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 65–76, Madison,Wisconsin, June 2002.

[22] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML Transformers. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 11–22, Dallas, Texas, May 15-17 2000.

[23] F. Neven. On the Power of Walking for Querying Tree-Structured Data. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 77–84, Madison, Wisconsin, June 2002.

[24] F. Neven and T. Schwentick. Query Automata. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 205–214, Philadephia, Pennsylvania, May 1999.

[25] F. Neven and T. Schwentick. Expressive and Efficient Pattern Languages for Tree-Structured Data. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 145–156, Dallas, Texas, May 2000.

[26] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Workshop on XML-Based Data Management (XMLDM) at the 8th Conference on Extending Database Technology*, pages 109–127, Prague, Mar. 2002. Springer-Verlag.

[27] ORACLE. ORACLE XML DB, January 2003. `http://otn.oracle.com/tech/xml/xmldb/pdf/XMLDB_Technical_Whitepaper.pdf%`.

[28] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 431–442, San Diego, California, June 2003.

[29] L. Segoufin. Typing and querying XML documents: some complexity bounds. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 167–178, San Diego, California, June 2003.

[30] L. Segoufin and V. Vianu. Validating Streaming XML documents. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 53–64, Madison, Wisconsin, June 2002.

[31] W3C XSL Working Group. XSL Transformations (XSLT) Version 2.0. W3C Working Draft, W3C, `http://www.w3.org/TR/xslt20/`, April 2002.

[32] X-HIVE. `http://www.x-hive.com`, 2002.