Efficient Algorithms for Evaluating XPath over Streams*

Gang Gou Rada Chirkova
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
ggou@ncsu.edu, chirkova@csc.ncsu.edu

ABSTRACT

In this paper we address the problem of evaluating XPath queries over streaming XML data. We consider a practical XPath fragment called Univariate XPath, which includes the commonly used '/' and '//' axes and allows $*$-node tests and arbitrarily nested predicates. It is well known that this XPath fragment can be efficiently evaluated in $O(|D||Q|)$ time in the non-streaming environment [18], where $|D|$ is the document size and $|Q|$ is the query size. However, this is not necessarily true in the streaming environment, since streaming algorithms have to satisfy stricter requirement than non-streaming algorithms, in that all data must be read sequentially in one pass. Therefore, it is not surprising that state-of-the-art stream-querying algorithms have higher time complexity than $O(|D||Q|)$.

In this paper we revisit the XPath stream-querying problem, and show that Univariate XPath can be efficiently evaluated in $O(|D||Q|)$ time in the streaming environment. Specifically, we propose two $O(|D||Q|)$-time stream-querying algorithms, LQ and EQ, which are based on the lazy strategy and on the eager strategy, respectively. To the best of our knowledge, LQ and EQ are the first XPath stream-querying algorithms that achieve $O(|D||Q|)$ time performance. Further, our algorithms achieve $O(|D||Q|)$ time performance without trading off space performance. Instead, they have better buffering-space performance than state-of-the-art stream-querying algorithms. In particular, EQ achieves optimal buffering-space performance. Our experimental results show that our algorithms have not only good theoretical complexity but also considerable practical performance advantages over existing algorithms.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems — Query Processing

General Terms: Algorithms, Theory.

Keywords: XML, XPath, Streams, Query Processing.

*This work was supported in part by NSF Career Award Grant 0447742 and NSF IIS Grant 0307072.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 12–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006 ...$5.00.

1. INTRODUCTION

There has been a growing practical need for querying XML data efficiently. In many emerging applications, such as monitoring stock market data, subscribing to real-time news, and managing network traffic information, XML data are available in *streaming* form only. An essential difference between querying streaming XML data and querying non-streaming XML data is that the former requires one-pass algorithms over unindexed XML data, in which all data have to be read sequentially and only once into memory.

1.1 Preliminaries

- **Data Model for XML Streams**

An XML document can be modeled as a rooted, labeled, and ordered tree, which we call XML *data tree*. Each node in the data tree corresponds to an element, attribute, or text value in the XML document. An XML streaming algorithm accepts input XML documents as a stream of SAX [7] events. Two core SAX events are $startElement(n)$ and $endElement(n)$, which are activated, respectively, when the opening or closing tag of a streaming element arrives, and accept the name of that element, n, as input parameter.

- **XPath**

XPath [13] is a popular language for querying XML data. It has been used in many XML applications and in some other languages for querying and transforming XML data, such as XQuery [6] and XSLT [12]. In this paper we address a practical fragment of XPath called Univariate XPath [3] (see Figure 1). A key characteristic of Univariate XPath is that in each Predicate, a Path can be compared with a Constant, but not with another Path. This paper does not treat explicitly the attribute axis @, since it can be handled in a way similar to the child axis /.

An XPath query specifies a twig pattern Q to navigate an XML data tree D. We denote the document size, i.e., the number of elements in D, by $|D|$, and the query size, i.e., the number of nodes in Q, by $|Q|$. There are three types of nodes in Q. There is exactly one *result node*, which specifies the output of XPath. All non-result nodes on the *main path* of Q, i.e., on the path from the root to the result node, are *axis nodes*. All other nodes are *predicate nodes*. Figure 2 shows the twig pattern of an XPath query Q: '//A[/E]/B[/F]//C[/G/I and //H]/D[/J]'[1], where the single-line edges represent /-axes, the double-line edges rep-

[1]To simplify the exposition, the examples given in this paper address queries with AND predicates only; note that our algorithms can handle predicates with mixed AND, OR, and NOT operators.

```
Path      :=  Step  |  Path Step
Step      :=  Axis NodeTest  |  Axis NodeTest  '[' Predicate ']'
Axis      :=  '/'  |  '//'
NodeTest  :=  name  |  '*'
Predicate :=  Path  |  Path CompOp Constant |
              Predicate 'and' Predicate | Predicate 'or' Predicate |
              'not' Predicate
CompOp    :=  '='  |  '!='  |  '>'  |  '>='  |  '<'  |  '<='
```

Figure 1: Grammar of Univariate XPath.

resent //-axes, and the result node 'D' is shaded. Further, we define four sub-patterns of Q w.r.t a given query node x in Q, as illustrated in Figure 2, where for $Q_{root}(x)$ and $Q_{down}(x)$, x can be any type of query node, and for $Q_{up}(x)$ and $Q_{self}(x)$, x is an axis node or result node (Section 4). We use these sub-patterns to explain our algorithms later.

In this paper we focus on addressing the *XPath querying problem*, which requires outputting all nodes in D (*answer nodes*) that satisfy a specified XPath twig pattern Q at its result node. We also address the *XPath filtering problem*, which requires determining whether there exists at least one match of Q in D. As we shall see in Section 3, stream-filtering algorithms could be used as a basis for stream-querying algorithms.

We define two usage-based memory-space classes for streaming algorithms: (1) *Caching space*, which is the memory space dynamically allocated for the run-time stack(s) (Section 2.1). (2) *Buffering space*, as required by stream-querying algorithms, which is the memory space dynamically allocated for temporarily storing potential answer nodes. We measure the size of buffering space as the maximal number of potential answer nodes buffered at a time during the running time, and denote it by B. B might reach $|D|$ in the worst case, which cannot be avoided by any stream-querying algorithms. A simple worst-case example is querying '//A[/B]/C' in an XML fragment of the form '$\langle a\rangle$ $\langle c_1\rangle$ 1 $\langle/c_1\rangle$ ...$\langle c_n\rangle$ n $\langle/c_n\rangle$ $\langle b\rangle$ 0 $\langle/b\rangle$ $\langle/a\rangle$', where c_1 through c_n have to be buffered until $\langle b\rangle$ arrives.

We categorize streaming algorithms into two classes based on when they evaluate the predicates in queries. We say that a streaming algorithm is *lazy* if it evaluates the predicates only when the closing tags of streaming elements are encountered, and is *eager* if it does that as soon as an atom in a predicate is evaluated to true. The eager strategy usually helps save buffering space significantly. A simple best-case example is querying '//A[/B]/C' in an XML fragment of the form '$\langle a\rangle$ $\langle b\rangle$ 0 $\langle/b\rangle$ $\langle c_1\rangle$ 1 $\langle/c_1\rangle$ ...$\langle c_n\rangle$ n $\langle/c_n\rangle$ $\langle/a\rangle$'. In the eager strategy, $B = 0$, since the predicate of a is evaluated (to true) as soon as $\langle b\rangle$ arrives. Thus, each c_i can be flushed as query result as soon as it arrives and does not need to be buffered at all. In the lazy strategy, $B = n$, since the predicate of a is not evaluated until $\langle/a\rangle$ arrives. Thus, all c_1 through c_n have to be buffered. We will compare these two strategies in more detail in Section 4.

- **Recursion in XML data**

Recursion, where some elements with the same name are nested on the same path in the data tree, occurs frequently in XML data in practice. For instance, among 60 DTDs surveyed in [11], 35 are recursive. [3] formally defines the *recursion depth* of an XML data tree D w.r.t query node q in Q, denoted by r_q, as the length of the longest sequence of nodes e_1, ..., e_{r_q} in D, such that (1) all the nodes lie on the same path, and (2) all the nodes match the sub-pattern $Q_{root}(q)$. It is easy to see that $r_q \leq d_D$, where d_D is the maximum depth of D. We define the recursion depth r of

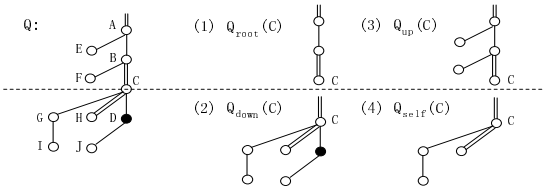


Figure 2: Sub-patterns of an XPath query Q.

D w.r.t Q as the maximum among all r_q's ($r \leq d_D$). For example, for the data tree and query in Figure 4, $r_A = r_B = 2$ and $r_C = r_D = r_E = 1$, and thus $r = 2$.

1.2 Related Work

- **Non-Streaming Algorithms**

A large amount of work has been done on efficiently querying indexed XML data, such as StackTree [1] and TwigStack [8], which improve query I/O performance by avoiding access to query-irrelevant data. These algorithms first pre-partition XML documents into multiple inverted lists and encode the position of each element, and then perform structural joins among the query-relevant inverted lists. These algorithms are not suitable for querying streaming data, since they require preprocessing or indexing XML data.

Another body of work addresses evaluating XPath over unindexed XML data, where the focus is on CPU performance. Gottlob et al. [18] identified an XPath fragment called Core XPath, which can be evaluated in $O(|D||Q|)$ time. Core XPath is slightly more expressive than Univariate XPath, in that it includes axes other than / and //. However, algorithms in [18] are not suitable for querying streaming data, since they require scanning XML documents in multiple passes. Papers by Gottlob et al. [19] and Segoufin et al. [29] study the theoretical complexity of evaluating XPath.

- **Streaming Algorithms**

Stream-Filtering Algorithms. Considerable work has been done in the context of filtering XML streams with a collection of XPath expressions. The key idea for improving the filtering performance is to share common sub-expressions among multiple XPath expressions, e.g., common path prefixes (YFilter [14]), common predicates (XPush [21]), or common substrings (XTrie [9]), as much as possible. Most of these algorithms are automata-based. YFilter [14] and the algorithm proposed in [20] mainly address linear path queries. For twig queries with predicates, YFilter uses an expensive post-processing step to join derived path tuples. XPush [21] supports twig queries with nested predicates, but the number of states in its automaton grows exponentially in $|Q|$ in the worst case, and thus it might incur exponential memory space costs. These automata-based algorithms are proposed in the context of stream filtering, and it is not clear how to extend them to efficient stream querying.

Stream-Querying Algorithms. Peng et al. [27] developed an eager automaton-based XPath stream-querying system XSQ. Similarly to XPush [21], the number of states in its automaton is exponential in $|Q|$ in the worst case. XSQ works in $O(|D| \cdot 2^{|Q|} \cdot k)$ time, where k is $O(r^{|Q|})$ in the worst case [10, 27]. XSQ might have to buffer multiple physical copies of a potential answer node at a time, since (due to the recursion in XML data streams) an answer node might have multiple matchings with the query. Bar-Yossef et al. [3] studied the theoretical caching-space

lower bound for evaluating XPath over streams, and also proposed a lazy stream-filtering algorithm S-Stack (Single-Stack). S-Stack uses a single run-time stack for caching, works in $O(|D| \cdot |Q| \cdot r)$ time, and uses $O(|Q| \cdot r)$ caching space for Univariate XPath. It lays a foundation for TurboXPath [22], a lazy stream-querying system for evaluating XQuery-like queries. More recently, Chen et al. [10] proposed a lazy stream-querying algorithm, TwigM, to avoid the exponential time and space complexity incurred by XSQ. TwigM extends the multi-stack framework of the TwigStack algorithm [8]. In [10], it is shown that TwigM can evaluate Univariate XPath in polynomial time and space in the streaming environment. Specifically, TwigM works in $O(|D||Q|(|Q| + d_D \cdot B))$ time and uses $O(|Q| \cdot r)$ caching space. However, like XSQ, TwigM might have to buffer multiple physical copies of a potential answer node at a time. XAOS [5], based on a single-stack framework similar to the framework of S-Stack, addresses evaluating XPath queries with both forward and backward axes. Bar-Yossef et al. [4] studied the theoretical buffering-space lower bound for evaluating Multi-Variate XPath over streams, and also proposed a stream-querying algorithm based on the eager strategy, which works for querying non-recursive XML streams only.

Some stream-querying systems for evaluating XQuery queries have recently been developed, such as BEA/XQRL [17], Flux [24], and XSM [25].

1.3 Motivations and Contributions

As we saw in Section 1.2, Univariate XPath can be evaluated in $O(|D||Q|)$ time in the non-streaming environment [18]. However, this excellent time performance has never been achieved by existing stream-querying algorithms. Although the time performance of evaluating Univariate XPath in the streaming environment has been improved from the exponential $O(|D| \cdot 2^{|Q|} \cdot r^{|Q|})$ of XSQ to the polynomial $O(|D||Q|(|Q|+d_D \cdot B))$ of TwigM, it has not yet reached the $O(|D||Q|)$ of the non-streaming environment. As we shall see in Section 5, the $|D||Q|$-independent time-complexity factors, such as $2^{|Q|} \cdot r^{|Q|}$ in XSQ and $d_D \cdot B$ in TwigM, could have a significant negative impact on the time performance of querying streaming XML data.

Therefore, a key question we ask here is: Could Univariate XPath be evaluated in $O(|D||Q|)$ time in the streaming environment? We observe that the $|D||Q|$-independent time-complexity factors of the existing stream-querying algorithms, such as $r^{|Q|}$ in XSQ and $d_D \cdot B$ in TwigM, result from the fact that these algorithms have to exhaustively visit a large number of (and sometimes almost all) elements cached in run-time stack(s) on the arrival of a streaming element in D. We believe that such nearly exhaustive exploration can be avoided. By carefully organizing and processing the elements cached in stacks and by carefully managing the buffered potential answer nodes in a multi-stack framework, we can obtain more efficient stream-querying algorithms that only need to visit a small portion of the relevant elements cached in stacks on the arrival of a streaming element in D, and thus achieve $O(|D||Q|)$ time performance. We now summarize our main contributions.

(1) We show that Univariate XPath can be efficiently evaluated in $O(|D||Q|)$ time in the streaming environment, using either the lazy strategy or the eager strategy. Specifically, we propose two $O(|D||Q|)$-time stream-querying algorithms, LQ and EQ, which are based on the lazy strategy and on the eager strategy, respectively. To the best of our knowledge, LQ and EQ are the first XPath stream-querying algorithms that achieve $O(|D||Q|)$ time performance.

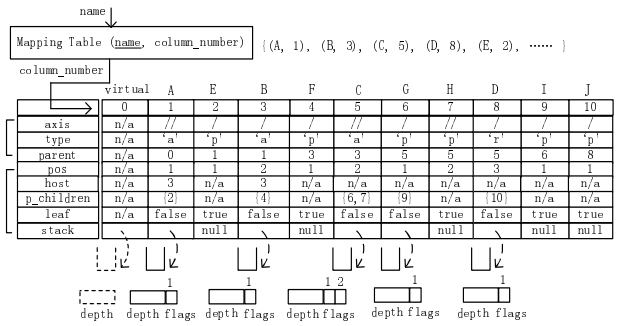


Figure 3: Query table for the query of Figure 2.

(2) We show that our algorithms are not only time-efficient but also space-efficient. Both LQ and EQ have the same ($|D|$-independent) caching-space complexity $O(|Q| \cdot r)$ as the existing state-of-the-art stream-querying algorithms. In particular, EQ tries to flush buffered query results out of memory as soon as possible, and achieves optimal buffering-space performance.

(3) Our extensive performance evaluation shows that our algorithms have not only good theoretical complexity but also considerable practical performance advantages over state-of-the-art stream-querying algorithms.

The rest of this paper is organized as follows. In Section 2 we propose a lazy stream-filtering algorithm LF, which lays a foundation for our lazy stream-querying algorithm LQ described in Section 3. In Section 4 we propose an eager stream-querying algorithm EQ. We report the results of an experimental performance evaluation in Section 5. We conclude in Section 6.

2. LAZY FILTERING ALGORITHM (LF)

We begin this section by introducing the query-preprocessing step (Section 2.1). In Section 2.2 we propose a basic LF algorithm, which is extended in Section 2.3 to a full LF algorithm that addresses queries with $*$-nodes (wildcards) or same-name nodes.

2.1 Query Preprocessing

A query table T_Q, illustrated in Figure 3, with size linear in $|Q|$, is statically stored in memory throughout stream processing. Each column in T_Q corresponds to a query node in Q. Column 0 represents a virtual node that is the parent of the root query node. A mapping table, implemented in the form of a hash table, is built over all (*name*, *column_number*) pairs for retrieving the *column_number* of any query node given the *name* of that node. We assume for now that Q has no $*$-nodes or same-name nodes.

T_Q includes the following fields for each query node c_n. (1) *axis*: / or //. (2) *type*: result node ('r'), axis node ('a'), or predicate node ('p'). This field can be ignored for the filtering problem. (3) *parent*, the column number of the parent node of c_n. (4) *pos*, which denotes the position of c_n among its sibling nodes. In Univariate XPath the order of sibling nodes is not significant, and we can select an arbitrary fixed order. (5) *host*, for the querying problem only (see Section 3.1). (6) *p_children*, the set of all predicate-node children of c_n. (7) f, the boolean formula defined w.r.t the predicate-node children of c_n. We do not explicitly record $f[c_n]$ in T_Q if AND is the only operator in $f[c_n]$ (e.g., for our example

Algorithm 1: B-LF::startElement(n)

```
1  depth=depth+1;
2  c_n = mapping(n); if c_n ≠ FAIL then LF::startBlock(c_n);
   Function mapping(n)
1  hash_id = hashing(n);
2  if MappingTable[hash_id].name = n then
3      return MappingTable[hash_id].column_number;
4  else return FAIL;
   Procedure LF::startBlock(c_n)
1  c_p = parent[c_n];      if stack[c_p] is empty then return;
2  p = top(stack[c_p]);
3  if p.flags[pos[c_n]] = 0 then
4      if axis[c_n] = '//' or p.depth+1 = depth then
5          if leaf[c_n] = true then
6              p.flags[pos[c_n]] = 1;
7          else
8              s = newElement(stack[c_n], depth);
9              push(stack[c_n], s);
```

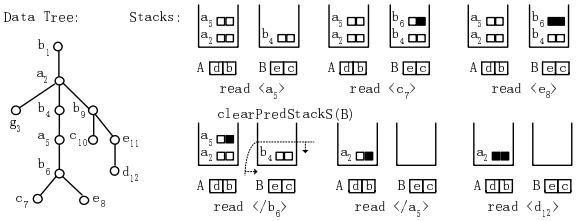


Figure 4: B-LF (query: '//A[//D]//B[/E]/C').

queries with AND predicates only). (8) $leaf$, which indicates whether c_n is a leaf query node. (9) *stack*, pointer to a run-time stack. In our algorithms, one run-time stack is created for each non-leaf query node.

Note that the full version of T_Q shown in Figure 3 is designed for the querying problem (Section 3). For the filtering problem we consider here, the *p_children* field of each axis node should also include its axis-node/result-node child, since all query nodes are viewed as predicate nodes in the filtering problem.

Each element in $stack[c_n]$ corresponds to an XML element e (data node) named n, and has two fields: (1) *depth*: the (integer) depth of e in the data tree. (2) $flags$: a bit array of size $|p\_children[c_n]|$. Given $c_m \in p\_children[c_n]$, $e.flags[pos[c_m]]$ indicates whether a match with $Q_{down}(c_m)$ has been found under e. Further, *flags* are partitioned into two groups, *cFlags* and *dFlags*, which correspond to / and // axis (predicate-node) children of c_n, respectively. We use the following stack functions: (1) $top(stack)$ returns the top element of *stack*. (2) $push(stack, element)$ pushes a new *element* into *stack*. (3) $pop(stack)$ pops out the top element of *stack*. (4) $destroy(element)$ recycles the memory space of a stack *element*. (5) $evaluate(element.flags, f[c_n])$ computes $f[c_n]$ based on the bits in $flags$ of a stack$[c_n]$ *element*.

2.2 The Basic LF Algorithm (B-LF)

Our basic LF algorithm (Algorithms 1 and 2), B-LF, addresses queries without $*$-nodes or same-name nodes. For brevity, we assume that no value comparisons are involved in predicates. It is straightforward to extend our algorithms to handle value comparisons.

Initially, an element v is pushed into $stack[0]$, the stack of the virtual query node at column 0, with $v.depth = 0$ and $v.flags[1] = 0$. The global variable *depth*, which denotes the depth of the streaming element being processed in the data tree, is initially set to 0, and is incremented/decremented by 1 in each startElement(n) / endElement(n) event.

Algorithm 2: B-LF::endElement(n)

```
1  c_n = mapping(n); if c_n ≠ FAIL then LF::endBlock(c_n);
2  depth = depth-1;
   Procedure LF::endBlock(c_n)
1  if leaf[c_n] = true or stack[c_n] is empty then return;
2  s = top(stack[c_n]);
3  if s.depth = depth then
4      pop(stack[c_n]);
5      if stack[c_n] is not empty then
6          q = top(stack[c_n]); q.dFlags = q.dFlags | s.dFlags;
7      if evaluate(s.flags, f[c_n]) = true then
8          c_p = parent[c_n];
9          if c_p = 0 then confirm that a match has been found;
10         else
11             p = top(stack[c_p]); p.flags[pos[c_n]] = 1;
12             if axis[c_n] = '//' then clearPredStackS(c_n);
13     destroy(s);
   Procedure clearPredStackS(c_n)
1  if stack[c_n] is not empty then
2      destroy all elements in stack[c_n];
3      for each c_i in p_children[c_n] do
4          if leaf[c_i] = false then clearPredStackS(c_i);
```

The basic idea of startBlock(c_n) is that an element qualifies for being pushed into $stack[c_n]$ (line 9) only if it has a match with $Q_{root}(c_n)$ (lines 1 and 4). $newElement(stack[c_n], depth)$ (line 8) creates a new $stack[c_n]$ element s with $s.depth = depth$, and initializes all bits in $s.flags$ to 0. The basic idea of endBlock(c_n) is evaluating $s.flags$ (line 7) to determine whether a match with $Q_{down}(c_n)$ has been found under p (line 11), as well as passing all those bits in $s.dFlags$ that have been set to 1 down to the new top element of $stack[c_n]$ (line 6). Note that line 12, which calls clearPredStackS(c_n) to clear all elements cached in $stack[c_n]$ and all descendant stacks of c_n, can be removed without impacting the correctness of B-LF. The cleared stack elements will not need to be evaluated when their closing tags arrive in future. The correctness of the B-LF algorithm is intuitive[2]. We now illustrate it using a running example.

Example 1. *Figure 4 shows a running example with several snapshots of stacks, where the depth numbers of the stack elements are not shown for simplicity. When $\langle g_3 \rangle$ is read, g_3 is simply discarded, since its tag name G is query-irrelevant (line 4 in mapping(n)). When $\langle /b_6 \rangle$ is read, b_6 is popped out of stack[B] (line 4 in endBlock(c_n)), and then $a_5.flags[b]$ is set to 1 and clearPredStackS(B) is called to destroy b_4 in stack[B] (lines 10-12 in endBlock(c_n)). When $\langle /a_5 \rangle$ is read, a_5 is popped out of stack[A], and $a_5.flags[b] = 1$ is passed down to $a_2.flags[b]$ (lines 5-6 in endBlock(c_n)). Note that b_9 does not need to be pushed into stack[B], since $a_2.flags[b] = 1$ (line 3 in startBlock(c_n)). Finally, when $\langle /a_2 \rangle$ is read, a match at a_2 is confirmed, since $a_2.flags[d] = 1$ and $a_2.flags[b] = 1$ (lines 7-9 in endBlock(c_n)).*

Recalling the definition of r in Section 1.1, we can see that r represents the maximal length of all run-time stacks during stream processing. Thus, the length of each stack is bounded from above by r. The caching-space complexity[3] of B-LF depends mainly on the number of $flags$ bits in

[2]Due to the space limit, in this paper we have to omit all rigorous correctness proofs.

[3]For clarity, in this paper we suppress trivial logarithmic factors in all space-complexity expressions. For example, strictly speaking, the *depth* field of each stack element takes $log(d_D)$ bits of memory space.

Algorithm 3: LF::startElement(n)

```
1 depth=depth+1;
2 c-sequence = mappingNameToColumns(n);
3 for each c_i in c-sequence in the c-sequence order do
4   LF::startBlock(c_i);

  Function mappingNameToColumns(n)
1 hash_id = hashing(n);
2 if MappingTable[hash_id].name = n then
3   return MappingTable[hash_id].column_sequence;
4 else return *-sequence;
```

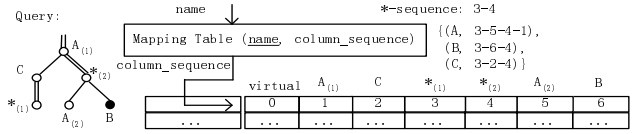


Figure 5: Mapping table (LF): *name → column sequence*.

all stack elements, which is $O(\sum_{c_i \in I(Q)} r \cdot fanout(c_i)) = O(r \cdot |Q|)$, where $I(Q)$ is the set of all non-leaf query nodes in Q. For the time complexity, startBlock(c_n) works in $O(max\{fanout(c_n), 1\})$, due to line 8 that initializes all bits in $s.flags$ to 0. endBlock(c_n) also works in $O(max\{fanout(c_n), 1\})$, due to line 6 that passes down $s.dFlags$, and to line 7 that evaluates $s.flags$. (Note that line 12, which can be removed without impacting the correctness of B-LF, has $O(1)$ amortized time cost for each cleared stack element, which will not need to be evaluated any more.) The bottleneck is the function $mapping(n)$, which takes only $O(1)$ time if the mapping table is a well-implemented hash table, or $O(|Q|)$ time if the mapping table is implemented as a naive sequential table. Thus, B-LF has time complexity $O(|D||Q|)$ (or $O(|D| \cdot F_Q)$ when using a well-hashed mapping table, where F_Q is the maximal fanout of Q ($F_Q \leq |Q|$)).

2.3 The Full LF Algorithm (LF)

It is easy to extend B-LF to address queries with $*$-nodes or same-name nodes. First, query preprocessing still creates one column for each query node. But each *name* in the mapping table corresponds to a sequence of column numbers whose corresponding query nodes either have that *name* or are $*$-nodes (see Figure 5). A special sequence for the column numbers of all $*$-nodes, called $*$-sequence, is also created. All nodes in column sequences follow a special order, such that each node must not have any of its ancestor nodes in front of itself. Here we implement this order using the post-order of nodes in the query twig. Our extended algorithm (Algorithms 3 and 4), called LF, iteratively calls LF::startBlock and LF::endBlock described in Algorithms 1 and 2.

The intuition behind LF is that when a streaming element named n arrives, LF visits all query nodes named n and all $*$-nodes. The order of visiting those nodes is crucial. As described in Algorithms 3 and 4, startBlock(c_i) should be called in the c-sequence order, and endBlock(c_i) should be called in the reverse c-sequence order, in order to prevent an element relevant to c_i from seeing its own copy in $stack[parent[c_i]]$. For example, in Case 1 in Figure 6, when $\langle a_1 \rangle$ is read, $a_1.flags[a]$ will be set to 1 if startBlock(A) is called after startBlock($*$), which will cause LF to eventually confirm an incorrect match at a_1. In Case 2 in Figure 6, when $\langle /b_3 \rangle$ is read, $d_1.flags[b]$ will miss being set to 1 if endBlock(B) is called before endBlock($*$), which will cause LF to eventually miss confirming a correct match at d_1.

Algorithm 4: LF::endElement(n)

```
1 c-sequence = mappingNameToColumns(n);
2 for each c_i in c-sequence in the reverse c-sequence order do
3   LF::endBlock(c_i);
4 depth=depth-1;
```

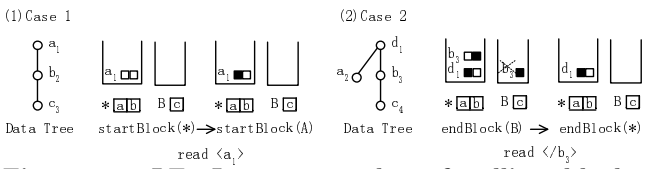


Figure 6: LF: Incorrect order of calling blocks (query: '//*[//A]/B/C').

As discussed in Section 2.2, both LF::startBlock(c_i) and LF::endBlock(c_i) work in $O(fanout(c_i))$ time if c_i is a non-leaf query node, and in $O(1)$ time otherwise. Since the length of each c-sequence is bounded by $|Q|$, both for-loops in Algorithms 3 and 4 work in time $O(\sum_{c_i \in I(Q)} fanout(c_i) + \sum_{c_i \in L(Q)} 1) = O(|Q|)$, where $L(Q)/I(Q)$ is the set of all leaf/non-leaf nodes in Q. Thus, Algorithms 3 and 4 work in $O(|Q|)$ time. Further, as discussed in Section 2.2, the length of each stack is bounded by r. Therefore, LF has the same caching-space complexity $O(|Q| \cdot r)$ as B-LF, although some elements might have multiple copies in different stacks. Note that in B-LF the sum of the lengths of all stacks is bounded by d_D, while in LF the lengths of all stacks may reach d_D at a time.

THEOREM 1. *For all queries in Univariate XPath, the LF algorithm correctly determines whether there exists a match of Q with D. It has time complexity $O(|D||Q|)$ and caching-space complexity $O(|Q| \cdot r)$.*

3. LAZY QUERYING ALGORITHM (LQ)

In this section, we extend our lazy stream-filtering algorithm LF to a lazy stream-querying algorithm LQ. LQ works in $O(|D||Q|)$ time and uses $O(|Q| \cdot r)$ caching space. That is, the time and caching-space complexity of LQ are not higher than those of LF, although the querying problem seems to be more complex than the filtering problem. We begin by introducing in Section 3.1 a query table, which is a slight extension of the table of Section 2.1. Next, in Section 3.2 we introduce algorithm U-LQ in which the answer nodes might be output not in the document order. In Section 3.3 we extend U-LQ to the full LQ algorithm that outputs answer nodes in the document order.

3.1 Query Preprocessing

In addition to the fields used in LF, the query table in LQ includes two more fields. (1) *type*, as described in Section 2.1. (2) *host*, for axis nodes only, which records the column number of the *segment host* of an axis node. Specifically, we partition the main path of Q into multiple segments by removing all // edges on it. The *host* of a segment is just the axis node at the tail of that segment. Meanwhile, we restrict the segment including the result node to not have a host. For example, for the query in Figure 2, its main path '//A/B//C/D' is partitioned into two segments: 'A/B' and 'C/D', where $host[A] = host[B] = B$, and C and D have no host.

Unlike LF, LQ requires buffering space for storing potential answer nodes, since LQ serves the querying purpose.

Algorithm 5: Procedure U-LQ::startBlock(c_n, id)

```
1  c_p = parent[c_n];      if stack[c_p] is empty then return;
2  p = top(stack[c_p]);
3  if type[c_n] ≠ 'p' or p.flags[pos[c_n]] = 0 then
4      if axis[c_n] = '//' or p.depth+1 = depth then
5          if leaf[c_n] = true then
6              if type[c_n] = 'p' then
7                  p.flags[pos[c_n]] = 1;
8              else if type[c_n] = 'r' then
9                  b = newBufferNode(id);
10                 if c_p = 0 then flushNode(b);
11                 else appendNode(p.list, b);
12         else
13             if type[c_n] = 'p' or type[c_n] = 'a' then
14                 s = newElement(stack[c_n], depth);
15             else s = newElement(stack[c_n], depth, id);
16             push(stack[c_n], s);
```

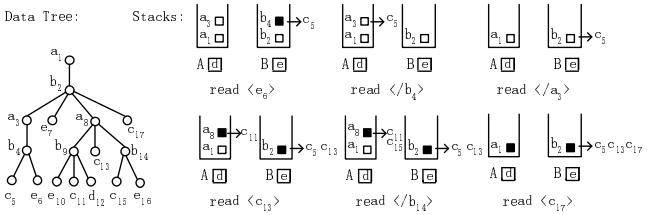


Figure 7: U-LQ (query: '//A[//D]/B[/E]//C').

Specifically, LQ creates two more fields for some stacks. (1) *list*, for the stacks of axis nodes only, which is a pointer to the head of a list that is used to buffer potential answer nodes. (2) *id*, for the stack of the result node only (if the result node is a non-leaf node), which records the id of a possible answer node.

3.2 The Unordered LQ Algorithm (U-LQ)

Similarly to Algorithms 3 and 4, U-LQ iteratively calls procedures startBlock and endBlock (Algorithms 5 and 6) in the c-sequence order and in the reverse c-sequence order, respectively. For XPath queries, an answer node might be output in the form of its text value, its unique node ID (if available), or the XML fragment rooted at it. Here for brevity we assume that the IDs of all answer nodes are available, and all answer nodes are output in the form of their IDs. It is easy to extend our algorithms to produce outputs in either of the other two forms.

It is easy to see that U-LQ shares substantial portions of code with LF, since U-LQ processes predicate nodes in the same way as LF does. (Recall that in LF all query nodes are considered predicate nodes.) However, U-LQ has extra code for processing axis nodes and result node. We define a *potential* answer node e as a data node that corresponds to the result node of Q and has been found to have a match with $Q_{down}(c_k)$, where c_k is some axis node or result node of Q. If c_k is the root node, then e is a real answer node. U-LQ creates *exactly one* physical copy for each potential answer node, and buffers it in the list of some stack element. U-LQ works in such a way that all nodes buffered in the list of a $stack[c_i]$ element, where c_i is an axis node, have been found to have a match with $Q_{down}(c_{i+1})$, where c_{i+1} is an axis node or result node with parent c_i.

U-LQ includes several additional functions not used in LF. (1) $newBufferNode(id)$ buffers a potential answer node in the form of its id. (2) $appendNode(list, node)$ appends a potential answer $node$ to the tail of $list$. (3) $appendList(list_1, list_2)$ appends $list_2$ to the tail of $list_1$. (4) $destroyList(list)$ recycles the memory space of all nodes in $list$. (5) $flushList(list)$ flushes all answer nodes in $list$ from memory to the user. (6) $flushNode(node)$ flushes an answer $node$ from memory to the user.

Algorithm 6: Procedure U-LQ::endBlock(c_n)

```
1  if leaf[c_n] = true or stack[c_n] is empty then return;
2  s = top(stack[c_n]);
3  if s.depth = depth then
4      pop(stack[c_n]);
5      if stack[c_n] is not empty then
6          q = top(stack[c_n]); q.dFlags = q.dFlags | s.dFlags;
7      if evaluate(s.flags, f[c_n]) = true then
8          c_p = parent[c_n];   p = top(stack[c_p]);
9          if type[c_n] = 'p' then
10             p.flags[pos[c_n]] = 1;
11             if axis[c_n] = '//' then clearPredStackS(c_n);
12         else if type[c_n] = 'a' then
13             if c_p = 0 then flushList(s.list);
14             else appendList(p.list, s.list);
15         else if type[c_n] = 'r' then
16             b = newBufferNode(s.id);
17             if c_p = 0 then flushNode(b);
18             else appendNode(p.list, b);
19     else
20         if type[c_n] = 'a' then
21             if stack[c_h = host[c_n]] is not empty then
22                 h = top(stack[c_h]);
23                 appendList(h.list, s.list);
24             else destroyList(s.list);
25     destroy(s);
```

Example 2. *Figure 7 shows a running example for U-LQ, which returns sequence $c_{11}c_{15}c_5c_{13}c_{17}$ as answer nodes. When $\langle c_5\rangle$ is read, c_5 is appended to $b_4.list$ (line 11 in Algorithm 5). When $\langle /b_4\rangle$ is read, $b_4.list = c_5$ is appended to $a_3.list$ (line 14 in Algorithm 6), since b_4's predicate is a successful match. When $\langle /a_3\rangle$ is read, the failure to match a_3's predicate causes $a_3.list = c_5$ to be appended to $b_2.list$, since $stack[B = host[A]]$ is not empty (lines 21-23 in Algorithm 6). When $\langle /a_8\rangle$ is read, the successful matching of a_8's predicate causes $c_{11}c_{15}$ in $a_8.list$ to be flushed to the user (line 13 in Algorithm 6). Finally, when $\langle /a_1\rangle$ is read, $c_5c_{13}c_{17}$ in $a_1.list$ are flushed to the user since a_1's predicate is a successful match.*

Now, suppose a_1's predicate fails to match when $\langle /a_1\rangle$ is read. Then, $c_5c_{13}c_{17}$ in $a_1.list$ will be destroyed, since $stack[B = host[A]]$ is now empty (line 24 in Algorithm 6).

Note that unlike the existing stream-querying algorithms, U-LQ does not need to exhaustively visit a large number of stack elements when a streaming element in D arrives. Specifically, startBlock(c_n, id) only needs to visit at most one stack element, p (line 2), and endBlock(c_n) only needs to visit at most three stack elements: s (line 2), q (line 6), and p (line 8) or h (line 22). That is, the time performance of U-LQ never depends on the stack length, i.e., on the recursion depth r. Therefore, compared to the existing stream-querying algorithms, a very nice property of U-LQ is that recursion in XML data streams has no impact on its time performance. Thus, U-LQ does not have the r-relevant time complexity factors of the existing stream-querying algorithms, such as $r^{|Q|}$ of XSQ and $d_D \cdot B$ of TwigM (Section 1.2).

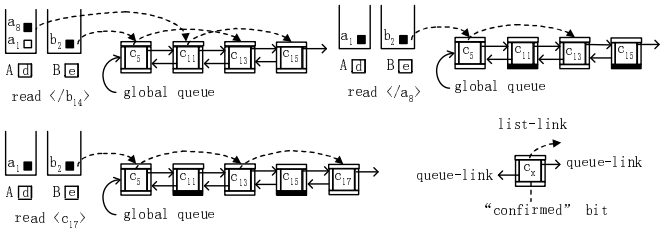


Figure 8: LQ (same query/data as in Figure 7).

Specifically, U-LQ has the same time complexity $O(|D||Q|)$ as LF. The $flushList(list)$ and $destroyList(list)$ functions have $O(1)$ amortized time cost for each node in $list$. In fact, after a potential answer node is buffered, it is visited only exactly once — when it is finally either flushed to the user by $flushList(list)$ or destroyed by $destroyList(list)$. The other additional functions, such as $appendList(list_1, list_2)$, take exactly $O(1)$ time. Also, it is easy to see that U-LQ has the same caching-space complexity $O(|Q| \cdot r)$ as LF. However, unlike LF, U-LQ requires buffering space for storing potential answer nodes in the lists of stack elements. It is easy to see that U-LQ uses no more buffering space than any lazy stream-querying algorithm. In particular, unlike TwigM and XSQ (Section 1.2), which might have to buffer multiple physical copies of a potential answer node at a time, U-LQ needs to buffer only exactly one physical copy for each potential answer node. Therefore, the buffering-space complexity of U-LQ is $O(|D|)$ in the worst case. (See the worst-case example in Section 1.1.)

3.3 The Full LQ Algorithm (LQ)

The LQ algorithm, which outputs answer nodes in the document order, is basically the same as U-LQ. The main difference is that LQ creates a global queue to collect all possible answer nodes in the document order. Specifically, in LQ each buffered node still has only one physical copy, but it might be linked into both the global queue and the list of some stack element, based on the double-link strategy illustrated in Figure 8. Also, each buffered node has an extra *confirmed* bit, which indicates whether this node has been confirmed to be an answer node or not. $flushList(list)$ (as well as $flushNode(node)$) in LQ does not flush answer nodes in $list$ to the user immediately as U-LQ does, since at this time some other nodes in the queue that are in front of the nodes in $list$ might have not been confirmed whether to be answer nodes or not. Thus, $flushList(list)$ in LQ just flips the $confirmed$ bits of the nodes in $list$ from 0 to 1, and then calls function $tryFlushingQueue()$, which tries to sequentially flush the nodes in the queue, beginning from the head of the queue, until a node with $confirmed = 0$ is encountered. $destroyList(list)$ in LQ unlinks those nodes in $list$ from the queue, recycles their memory space, and then calls $tryFlushingQueue()$. The reason is, after the removal of those nodes in $list$ from the queue, some answer nodes in the queue that have been confirmed earlier might be able to reach the head of the queue, and thus can be flushed to the user immediately.

EXAMPLE 3. *Figure 8 shows a running example for LQ. When $\langle/a_8\rangle$ is read, $c_{11}c_{15}$ in $a_8.list$ are confirmed as answer nodes, and have their $confirmed$ bits flipped to 1 by $flushList(a_8.list)$. But the head of the queue, c_5, still has $confirmed = 0$. Thus, no answer nodes can be flushed to the user at this time. All answer nodes will be flushed only when $c_5c_{13}c_{17}$ in $a_1.list$ have their confirmed bits flipped to 1 by $flushList(a_1.list)$ when $\langle/a_1\rangle$ arrives.*

Now, suppose b_2's predicate fails to match, then $c_5c_{13}c_{17}$ in $b_2.list$ would be unlinked from the queue by $destroyList$ ($b_2.list$) when $\langle/b_2\rangle$ is read. As a result, c_{11} and c_{15} can be flushed to the user immediately, since c_{11} has reached the head of the queue.

Since the global queue collects nodes in the document order, all answer nodes output by LQ are in the document order. It is easy to see that while LQ does not have extra time or space complexity compared to that of U-LQ, in practice LQ might use more buffering space than U-LQ, since LQ might have to delay flushing some confirmed answer nodes in order to output the answer nodes in the document order.

THEOREM 2. *For all queries in Univariate XPath, the LQ algorithm correctly outputs all answer nodes in the document order. It has time complexity $O(|D||Q|)$, caching-space complexity $O(|Q|\cdot r)$, and buffering-space complexity $O(|D|)$.*

Theorem 2 shows that Univariate XPath can be efficiently evaluated in the same $O(|D||Q|)$ time in the streaming environment as in the non-streaming environment [18].

4. EAGER QUERYING ALGORITHM (EQ)

Although LQ achieves $O(|D||Q|)$ time performance, it does not reach optimal buffering-space performance. It can be seen from Figure 7 that, once d_{12} arrives, there is enough information to confirm buffered nodes, c_5 and c_{11}, as answer nodes. Thus, they can be flushed to the user immediately. Also, c_{13}, c_{15}, and c_{17} can be confirmed as answer nodes and be flushed as soon as they arrive. That is, the size of buffering space can be reduced from 4 nodes in U-LQ (or from 5 nodes in LQ) to 2 nodes. The reason for the lower buffering-space performance of LQ is that it lazily evaluates the predicates only when the closing tags of the streaming elements are encountered (line 7 in Algorithm 6). Motivated by this, in this section we propose an eager stream-querying algorithm *EQ*, which improves buffering-space performance, while not trading off time performance.

4.1 Query Preprocessing

The query table for EQ includes two extra fields: (1) $axis\_child[c_n]$, for axis nodes only, which records the column number of the axis-node/result-node child of c_n. (2) $PF[c_n]$ (predicate fanout), which is the size of $p\_children[c_n]$.

EQ uses four more fields for some stacks. (1) the *parent* pointer. Given a $stack[c_n]$ element e, $e.parent$ is a pointer to the closest ancestor of e among all $stack[parent[c_n]]$ elements. The left arrows between stacks in Figure 9 illustrate parent pointers. (2) the *child* pointer, for the stacks of axis nodes only. Given a $stack[c_n]$ element e', $e'.child$ points to the closest descendant, say e, of e' among all $stack[axis\_child[c_n]]$ elements such that $e.parent = e'$, and is $NULL$ if such e does not exist. The right arrows between stacks in Figure 10 illustrate child pointers. For example, $b_{14}.child = c_{15}$ while $b_{12}.child = NULL$ and $b_{10}.child = NULL$, since $c_{15}.parent = b_{14}$. Note that in Figures 9 and 10, all stack elements come from the same path in a data tree, and the subscript number of each stack element indicates (but is not exactly[4]) the depth of that element in the data tree. For simplicity, we do not show parent pointers in

[4]Note that for simplicity, we do not show all stacks of the query.

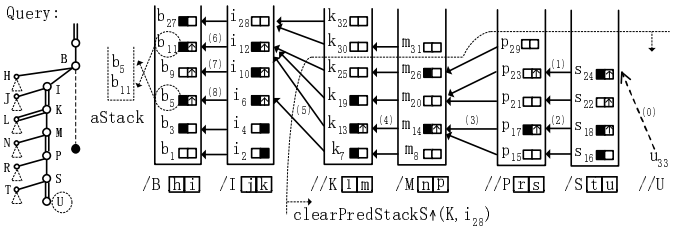


Figure 9: U-EQ: bottom_up_Evaluate(U).

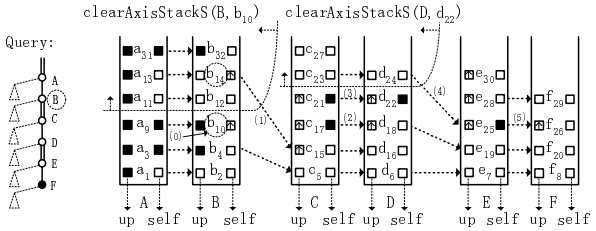


Figure 10: U-EQ: top_down_Propagate($b_{10}b_{14}$).

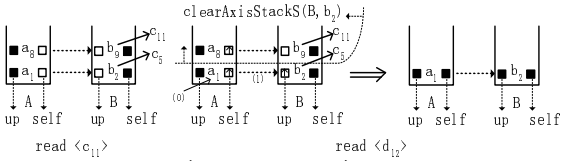


Figure 11: U-EQ (same query/data as in Figure 7).

Figure 10; they can be inferred from the subscript numbers of stack elements. (3)-(4) the *self* and *up* bits, for the stacks of axis nodes and result node only. Given a $stack[c_n]$ element e, $e.self$ indicates whether e has a match with $Q_{self}(c_n)$, i.e., whether $e.flags$ has been evaluated to true, and $e.up$ indicates whether e has a match with $Q_{up}(c_n)$, where $Q_{self}(c_n)$ and $Q_{up}(c_n)$ are as illustrated in Figure 2.

4.2 Algorithm

We first describe our U-EQ algorithm, in which answer nodes might be output not in the document order. Similarly to Algorithms 3 and 4, U-EQ iteratively calls procedures startBlock and endBlock (Algorithms 7 and 8) in the c-sequence order and in the reverse c-sequence order, respectively. Here we address queries with AND predicates only. It is straightforward to extend U-EQ to address queries with the AND, OR and NOT operators.

The main work of U-EQ is lines 7-8 in Algorithm 7. In line 7, $bottom\_up\_Evaluate(c_n)$ (BUE) eagerly evaluates a predicate as soon as an atom in that predicate becomes true. BUE works in a bottom-up way. Figure 9 illustrates this process, in which the ↑ symbol in a white square indicates that that bit is being flipped from 0 to 1 by this BUE call, and the left arrows with numbers indicate the *go-forward* path of this BUE call. Also, the path from node B to node S in the query is logically partitioned into three segments by removing all // edges on that path. The main idea of BUE is that given a $stack[c_n]$ element e, BUE evaluates $e.flags$ as soon as a corresponding bit in $e.flags$ is flipped to 1. If $e.flags$ is evaluated to true (in this case we say that e becomes *activated*), then BUE *goes forward* to $e.parent$ and flips $e.parent.flags[pos[c_n]]$ to 1. Otherwise, BUE has to *go down* to the lower segment-tail element. In Figure 9, u_{33} flips $s_{24}.flags[u]$ to 1, which activates s_{24}. Thus, BUE goes forward to p_{23}. But p_{23} cannot be activated, and thus BUE has to go down to s_{22}. The remaining process follows similar rules. When the axis-node elements b_{11} and b_5 become activated, their $self$ bits are flipped to 1, and then they are pushed into $aStack$ that is eventually returned by this BUE call. Finally, $clearPredStackS_{\uparrow}(K, i_{28})$ is called, which destroys all those $stack[K]$ elements whose depth is smaller than $i_{28}.depth$, and then recursively calls $clearPredStackS_{\uparrow}(M, k_{30})$ and $clearPredStackS_{\uparrow}(L, k_{30})$. BUE terminates as soon as it finds that $i_4.flags[k]$ has been set to 1 earlier (which implies that no more elements will become activated even if BUE continues).

In line 8, $top\_down\_Propagate(aStack)$ (TDP) processes the axis-node elements returned by BUE (whose $self$ bits were just flipped to 1 in BUE), as illustrated in Figure 10[5]. If the *up* bits of those elements have been set to 1, then TDP will try flipping the *up* bits of their corresponding descendant elements to 1 in a top-down way. Specifically, given a $stack[c_n]$ element e, TDP checks the value of $e.self$ as soon as $e.up$ is flipped to 1. If $e.self$ has been set to 1 (in this case we say that e becomes *activated*), then TDP *goes forward* to $e'.child$ in $stack[axis\_child[c_n]]$ and flips $e'.child.up$ to 1, where e' is the lowest element in $stack[c_n]$ that is e itself or e's descendant with $e'.child \neq NULL$. Otherwise, BUE has to *go up* to the upper segment-head element. In Figure 10, initially, b_{10} is activated. Then TDP goes forward to $b_{14}.child$, c_{15}, since $b_{10}.child$ and $b_{12}.child$ are NULL, before $clearAxisStackS(B, b_{10})$ is called to destroy those $stack[A]$ and $stack[B]$ elements whose depth is greater than b_{10}.depth. Meanwhile, all nodes buffered in the lists of the cleared elements and of b_{10} can be flushed immediately as answer nodes, since (1) b_{10} has matches with $Q_{up}(B)$ and $Q_{self}(B)$, and (2) these buffered nodes have matches with $Q_{down}(C)$. Also, $clearPredStackS_{\downarrow}(A, a_{11})$ and $clearPredStackS_{\downarrow}(B, b_{12})$ are called to recursively destroy the corresponding upper elements in the stacks of the descendant predicate nodes of A and B, based on $a_{11}.depth$ and $b_{12}.depth$, respectively. Next, $c_{15}.up$ is flipped to 1. But c_{15} cannot be activated, and thus TDP goes up to c_{17}. The remaining process follows similar rules. Note that TDP will terminate as soon as it reaches an element whose *up* bit has been set to 1 earlier (which implies that no more elements will become activated even if TDP continues).

EXAMPLE 4. *Figure 11 shows a running example for U-EQ. When $\langle d_{12} \rangle$ is read, $BUE(D)$ (line 7 in Algorithm 7) is called and returns a_1a_8. Then, $TDP(a_1a_8)$ is called (line 8 in Algorithm 7): a_1 becomes activated first, and then b_2 becomes activated. Finally, a_8 and b_9 are cleared, and c_5 in $b_2.list$ and c_{11} in $b_9.list$ are flushed as answer nodes. Later, when c_{13}, c_{15}, and c_{17} arrive, they are flushed immediately, since $b_2.up = 1$ and $b_2.self = 1$ (line 11 in Algorithm 7). As a result, throughout the processing, U-EQ only needs to buffer at most 2 nodes, c_5 and c_{11}, rather than 4 nodes in U-LQ and 5 nodes in LQ (see Examples 2 and 3).*

Now, suppose that edge (b_2, c_{17}) in the data tree in Figure 7 is extended to a path b_2-a_x-b_y-c_{17}; a_x and b_y never need to be pushed into stacks, because when a_x arrives, $b_2.up = 1$ and $b_2.self = 1$ (lines 17-19 in Algorithm 7). Thus, c_{17} will still be flushed as soon as it arrives.

Note that U-EQ::startBlock(c_n, *id*) seems to have particularly high time costs, since both BUE and TDP might visit a lot of elements in stacks. However, such costs are amor-

[5]Here we assume that the six $stack[B]$ elements in Figure 10 correspond to the six $stack[B]$ elements in Figure 9, respectively.

Algorithm 7: Procedure U-EQ::startBlock(c_n, id)

```
1  c_p = parent[c_n];      if stack[c_p] is empty then return;
2  p = top(stack[c_p]);
3  if type[c_n] ≠ 'p' or p.flags[pos[c_n]] = 0 then
4      if axis[c_n] = '//' or p.depth+1 = depth then
5          if leaf[c_n] = true then
6              if type[c_n] = 'p' then
7                  aStack = bottom_up_Evaluate(c_n);
8                  top_down_Propagate(aStack);
9              else if type[c_n] = 'r' then
10                 b = newBufferNode(id);
11                 if p.up & p.self = 1 then flushNode(b);
12                 else appendNode(p.list, b);
13         else
14             if type[c_n] = 'p' then
15                 s = newElement(stack[c_n], depth);
16             else if type[c_n] = 'a' then
17                 if stack[c_h = host[c_n]] is not empty then
18                     h = top(stack[c_h]);
19                     if h.up & h.self = 1 then return;
20                 s = newElement(stack[c_n], depth);
21             else s = newElement(stack[c_n], depth, id);
22             s.parent = p;
23             if type[c_n] = 'a' or type[c_n] = 'r' then
24                 s.up = p.up & p.self;
25                 if PF[c_n] = 0 then s.self = 1;
26                 else s.self = 0;
27                 s.child = NULL;
28                 if p.child = NULL then p.child = s;
29             push(stack[c_n], s);
```

tized when the closing tags of these stack elements are processed in future: U-EQ::endBlock(c_n) works in $O(1)$ time, since it does not need to do the work of passing / evaluating $s.flags$ as lines 6-7 in Algorithm 6 do[6]: all such work has been transferred into BUE in U-EQ::startBlock(c_n, id).

Further, the key point for BUE is that during the lifetime of each $stack[c_n]$ element e, each bit in $e.flags$ is flipped from 0 to 1 at most once (by some BUE call), since BUE terminates as soon as it finds that the corresponding bit in $e.flags$ that it is trying to flip to 1 has been set to 1 earlier. Each such flip action causes $e.flags$ to be evaluated once, to test whether e will be activated. That is, $e.flags$ is evaluated at most $PF[c_n]$ times. We can create a *counter* field for each stack element e, to record the number of bits in $e.flags$ that have been set to 1. The value of $e.counter$ is initialized to 0 and is incremented by 1 every time a bit in $e.flags$ is flipped to 1. Then, $e.flags$ can be evaluated in $O(1)$ time by simply checking whether $e.counter = PF[c_n]$. Therefore, on the whole, each $stack[c_n]$ element takes only $O(max\{PF[c_n], 1\})$ time for the BUE computation during its lifetime, although a BUE call may visit many stack elements. Similarly, each axis-node/result-node stack element e takes $O(1)$ time for the TDP computation during its lifetime, since $e.up$ is flipped from 0 to 1 at most once. (Recall that TDP terminates as soon as it finds that an *up* bit that it is trying to flip to 1 has been set to 1 earlier.) Otherwise, as in LF and LQ, all *clearStackS* functions in BUE and TDP have $O(1)$ amortized time cost for each cleared stack element, which does not need to be processed further in future. Therefore, on the whole, each stack element has $O(max\{PF[c_n], 1\})$ time cost in U-EQ, as in LQ.

[6]In addition, line 14 in U-EQ::endBlock(c_n) has the $O(1)$ amortized time cost, as discussed in Section 3.2.

Algorithm 8: Procedure U-EQ::endBlock(c_n)

```
1  if leaf[c_n] = true or stack[c_n] is empty then return;
2  s = top(stack[c_n]);
3  if s.depth = depth then
4      pop(stack[c_n]);
5      if type[c_n] = 'a' or type[c_n] = 'r' then
6          c_p = parent[c_n];   p = top(stack[c_p]);
7          if p.child = s then p.child = NULL;
8          if type[c_n] = 'a' then
9              if s.self = 1 then appendList(p.list, s.list);
10             else
11                 if stack[c_h = host[c_n]] is not empty then
12                     h = top(stack[c_h]);
13                     appendList(h.list, s.list);
14                 else destroyList(s.list);
15     destroy(s);
```

It is easy to see that U-EQ has the same caching-space complexity $O(|Q| \cdot r)$ as LQ. Also, similarly to LQ, U-EQ has $O(|D|)$ buffering-space complexity in the worst case. Such complexity is unavoidable for any stream-querying algorithm. (See the worst-case example in Section 1.1.) However, unlike LQ, U-EQ is able to save buffering space significantly, as illustrated by the simple best-case example in Section 1.1 and by Example 4. In fact, U-EQ achieves optimal buffering-space performance, since it always flushes confirmed answer nodes out of memory as soon as possible, and never buffers any answer nodes unnecessarily. Further, similarly to the procedure of Section 3.3, we can extend U-EQ to a full EQ algorithm, called EQ, which outputs answer nodes in the document order by using a global queue. This does not incur extra complexity over that of U-EQ.

THEOREM 3. *For all queries in Univariate XPath, the EQ algorithm correctly outputs all answer nodes in the document order. It has time complexity $O(|D||Q|)$, has caching-space complexity $O(|Q| \cdot r)$, and achieves optimal buffering-space performance.*

Theorem 3 shows that for the problem of evaluating Univariate XPath over streams, optimal time performance *and* optimal space performance can be achieved at the same time.

5. EXPERIMENTAL RESULTS

5.1 Experimental Setup

In this section we compare the performance of our algorithms with that of two state-of-the-art XPath stream-querying systems, TwigM [10] and XSQ [27]. These two systems, in particular TwigM, have shown comprehensive performance advantages over many stream-querying systems [10]. Other XPath or XQuery stream-querying systems, such as BEA/XQRL [17], TurboXPath [22], XAOS [5] and XSM [25], are not publicly available at this time, while some publicly available XPath or XQuery querying systems, such as Galax [16], XMLTaskForce [18] and Saxon [23], use non-streaming algorithms. XSQ is an open-source system [26], while TwigM is not publicly available at this time; we implemented it based on the algorithm described in [10]. All the algorithms were implemented using Java 1.4.2, and called the same SAX XML parser (Xerces2-Java XML Parser 2.8.1 [2]). We ran all the experiments on a 2GHz Pentium4 machine with 2GB memory running Windows Server 2003.

	Book	Treebank	XMark
Data size	12M	82M	113M
Number of nodes	114K	2437K	1666K
Max/Avg depth	22/19.4	36/7.8	12/5.5

Figure 12: Datasets for the experimental evaluation.

Book dataset
Q1: //book/section/figure/image
Q2: //section[./figure]//title
Q3: //section[./title]//figure
Q4: //section[./figure/image]/section[./title]//figure
Q5: //section[./figure and ./title]/*[.//image]//title

Treebank dataset
Q6: //VP//NP//NN
Q7: //S[./NP]//VP//NN
Q8: //S[./PP]//NP//NNP
Q9: //VP[./PP/IN]//NP[]/*
Q10: //S[./CC and /PP]//NP[.//VBZ and ./IN]//JJ

XMark dataset
Q11: //person[./creditcard]//business
Q12: //people/person[./profile]//interest
Q13: //open_auction[./initial]/bidder/personref
Q14: //open_auctions/open_auction[./privacy]/*
Q15: //item[.//payment and .//shipping]//mailbox/mail[./date]/to

Figure 13: Queries for the experimental evaluation.

We tested the performance of all algorithms on three datasets (Figure 12): (1) the *Book* dataset, which is a synthetic dataset generated using IBM's XML Generator [15] (with $NumberLevels = 20$ and $MaxRepeats = 9$), based on a real *Book* DTD from W3C XQuery Use Cases [31]. The Book DTD includes only one recursive element, *section*. That is, different *section* nodes can be nested on the same path in the data tree. (2) the *Treebank* dataset [30], which is a real dataset with a narrow and deeply recursive structure that includes multiple recursive elements. (3) the *XMark* dataset (with $factor = 1$) [28], which is a well-known benchmark dataset. This dataset does not include recursive elements.

On each of the three datasets, we tested 5 queries, as shown in Figure 13. These queries include / and // axes, $*$-node tests, and predicates. We used queries with AND operators only, since neither TwigM nor XSQ support other operators, and XSQ even does not support AND operators. Further, XSQ does not support same-name nodes or $*$-node tests, and requires that each axis node have at most one (/-axis) predicate node child. Thus, some queries listed in Figure 13 are not supported by XSQ, and we use N/A to indicate such cases in Figures 14 and 15.

5.2 Time Performance

We tested the CPU time performance of the stream-querying algorithms, measuring it as $T = t_{total} - t_{in} - t_{out}$, where t_{total} is the total running time, t_{in} is the time taken by reading (from disk into memory) and parsing XML documents, and t_{out} is the time taken by outputting the query results from memory to disk. Figure 14 shows the CPU time performance of our algorithms, of TwigM, and of XSQ on the three datasets. From this figure we can see that EQ and LQ have the best time performance among all the algorithms. This performance advantage is rather intuitive: As shown in Theorems 2 and 3, both LQ and EQ have $O(|D||Q|)$ time complexity, and thus other factors, such as recursion in XML data or the size of buffering space, have no impact on their time performance. We now summarize several main observations based on Figure 14.

(1) XSQ has much higher time costs than the other algorithms in all applicable test cases. This is intuitive, since XSQ has exponential time complexity $O(|D| \cdot 2^{|Q|} \cdot r^{|Q|})$ (Section 1.2), while EQ, LQ, and TwigM each have polynomial time complexity. In particular, in presence of deep recursion in XML data, e.g., when evaluating $Q7$ or $Q8$ on the Treebank dataset (Figure 14 (b)), XSQ reports "there are too many path combinations for one element", and terminates running. The reason is, XSQ exhaustively enumerates all potential main-path pattern matches for each potential answer node, and the number of such matches might reach $r^{|Q|}$ in the worst case [10].

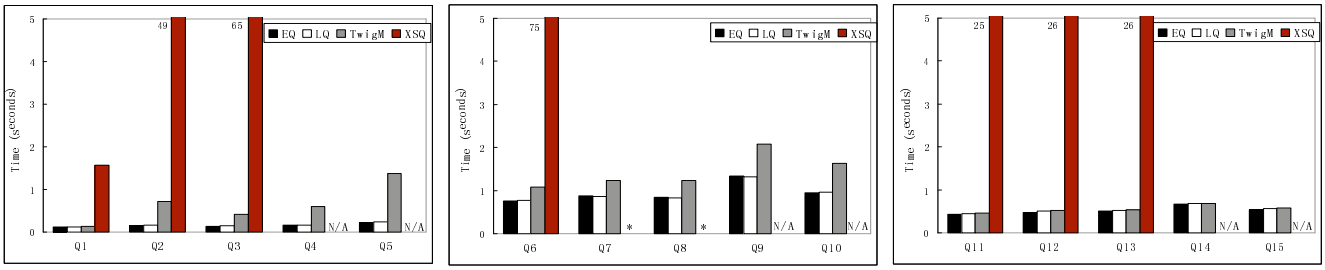
(2) EQ and LQ show different degrees of time-performance advantages over TwigM on different datasets.

On the Book dataset, EQ and LQ have a marked performance advantage over TwigM for $Q2$ through $Q5$. As we can observe from the time complexity $O(|D||Q|(|Q| + d_D \cdot B))$ of TwigM (Section 1.2), the size of the run-time buffering space has a significant impact on the time performance of TwigM. Figure 15 (a) shows the buffering-space size of TwigM for $Q1$ through $Q5$ on the Book dataset. From this figure we can see that TwigM has high buffering-space costs for Q_2 through Q_5, which results in its high time costs for these queries. Note that $O(|D||Q|(|Q| + d_D \cdot B))$ serves only as the theoretical upper bound of the time complexity of TwigM. It does not imply that each streaming element in D requires $O(B)$ processing time. In fact, there might be only a few query-relevant streaming elements that require $O(B)$ processing time (for the expensive set-union operation [10] that is used to eliminate duplicate copies of buffered nodes). In particular, most query-irrelevant streaming elements (such as g_3 in Figure 4) can be simply discarded in $O(1)$ time by hashing. On the other hand, TwigM's time performance is similar to that of EQ and LQ on $Q1$, since the Book dataset is not recursive w.r.t $Q1$ (the axis of *section* in $Q1$ is /). When there is no recursion, TwigM does not need to buffer duplicate copies of potential answer nodes, and thus does not incur the $O(B)$ set-union costs anymore. That is, in non-recursive cases, the size of the run-time buffering space has no impact on the time performance of TwigM. Thus, TwigM has the same $O(|D||Q|)$ time complexity as EQ and LQ in non-recursive cases.

On the Treebank dataset, the performance advantage of EQ and LQ over TwigM is not as pronounced as on the Book dataset. The reason is, while Treebank is deeply recursive, it is a very narrow dataset, on which TwigM has very low buffering space costs, as indicated by Figure 15 (b). Despite this, EQ and LQ still have a performance advantage over TwigM on this dataset, mainly due to the impact of the deeply recursive structure of this dataset on the time performance of TwigM.

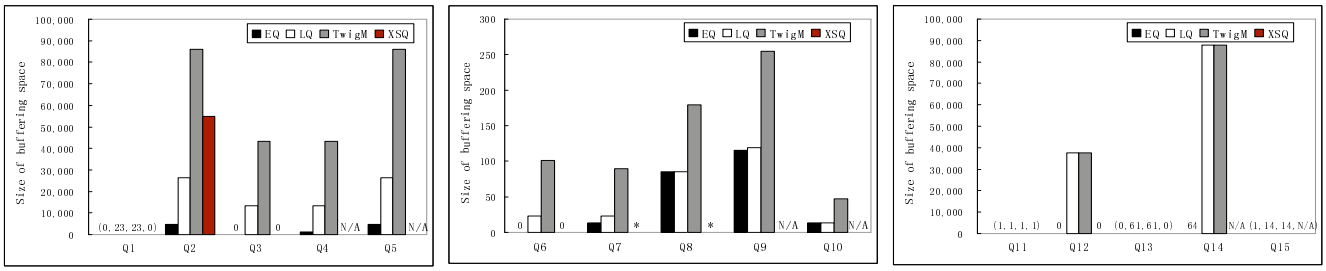
On the XMark dataset, TwigM's time performance is very similar to that of EQ and LQ, since XMark is a non-recursive dataset. As discussed above, TwigM has $O(|D||Q|)$ time complexity in non-recursive cases.

(3) EQ and LQ show stable time performance for different queries in each fixed dataset, except for some queries involving $*$-node tests. For example, in Figure 14 (b), EQ and LQ have markedly higher costs for $Q9$ than for other queries. The reason is, for such queries all streaming elements become query-relevant (w.r.t. $*$-nodes). Thus, no elements can be simply discarded during the stream-querying process.



(a) Book dataset (b) Treebank dataset (c) XMark dataset

Figure 14: Query time[7].



(a) Book dataset (b) Treebank dataset (c) XMark dataset

Figure 15: Size of buffering space (number of nodes)[7].

(4) Our two $O(|D||Q|)$-time stream-querying algorithms, EQ and LQ, show very similar time performance in practice, as can be seen from Figure 14. The maximum and average difference between their time performance is 7% and 3%, respectively.

5.3 Memory Space Performance

The caching-space costs of stream-querying algorithms depend on the number of elements cached in the run-time stack(s), which is bounded by the maximum document depth d_D when queries do not involve $*$-nodes or same-name nodes, and does not exceed $|Q| \cdot d_D$ in the worst case. Many practical XML documents are not very deep. Thus, the caching-space costs of stream-querying algorithms are almost always negligible in practice. As we observed in our test cases, the number of stack elements cached by each algorithm does not exceed 40 on the Book dataset and 25 on the Treebank dataset at any running time, and in particular, is at most 3 on the XMark dataset, since each stack only needs to cache at most one corresponding element in non-recursive cases. On the other hand, stream-querying algorithms could typically have very high buffering-space costs in practice, since hundreds of thousands of potential answer nodes might have to be buffered, see Figures 15 (a) and (c). Therefore, the run-time memory usage of stream querying is typically dominated by buffering space. In Figure 15 we compare the buffering-space size of EQ, LQ, TwigM, and XSQ[7]. Note that TwigM and XSQ might buffer multiple physical copies of a potential answer node at the same time. Thus, their buffering-space size is measured using the maximal number of physical copies of potential answer nodes buffered at the same time during the running time. We now summarize several main observations based on Figure 15.

[7]'N/A' means that XSQ does not support this query type, while '$*$' means that XSQ reports "there are too many path combinations for one element" due to the deeply recursive structure of Treebank, and terminates running.

(1) Buffering-space costs could be very large in practice, depending on the structure of XML data, on the specific queries, and on the query-evaluation strategy. For example, for Q14 in Figure 15, the lazy algorithms LQ and TwigM have to buffer almost 90K nodes. The reason is, the *open_auctions* element in the XMark dataset has a very large number of *open_auction* child elements, most of which have a *privacy* child element. Therefore, $Q14$ has a very large number of answer nodes in this dataset. The lazy strategy has to buffer all these answer nodes until $\langle$/*open_auctions*$\rangle$ arrives. It is easy to see that there will be a lot more (than 90K) nodes that need to be buffered when $Q9$ is lazily evaluated on other larger XMark datasets (with higher value of the $factor$ parameter). On the other hand, queries on the Treebank dataset need very little buffering space, since Treebank is a narrow dataset.

(2) EQ always has the lowest buffering-space costs among all the algorithms. This is intuitive; as we discussed in Section 4, EQ always flushes buffered answer nodes as soon as possible, and thus achieves optimal buffering-space performance. Although XSQ also eagerly evaluates queries, it sometimes has to buffer multiple physical copies of some potential answer nodes. Thus, XSQ could have higher buffering-space costs than EQ in some cases, e.g. for Q2 in Figure 15 (a). Both LQ and TwigM use the lazy strategy to evaluate queries, and therefore have higher buffering-space costs than EQ in most test cases. However, compared to LQ, TwigM might have to sometimes buffer multiple physical copies of some potential answer nodes. Thus, TwigM typically has higher buffering-space costs than LQ, see Figures 15 (a) and (b). Note that compared to other algorithms, the amount of buffering space EQ saves could be very large. For example, for Q14 in Figure 15 (c), EQ only needs to buffer at most 64 nodes, while LQ and TwigM have to buffer almost 90K nodes.

(3) In non-recursive cases, EQ and XSQ have the same buffering-space costs, while LQ and TwigM have the same buffering-space costs, as shown in Figure 15 (c). The reason

is, both EQ and XSQ use the eager strategy to evaluate queries, while both LQ and TwigM use the lazy strategy to evaluate queries. In non-recursive cases, XSQ and TwigM do not have to buffer multiple physical copies of potential answer nodes.

5.4 Summary

Our experiments illustrate two points.

(1) Our algorithms EQ and LQ show the best time performance among all the tested algorithms in practice, which is consistent with their good $O(|D||Q|)$ worst-case time complexity. In presence of recursion, they have a marked time-performance advantage over the state-of-the-art stream-querying algorithm TwigM. At the same time, they guarantee time performance that is similar to that of TwigM in the absence of recursion.

(2) EQ shows the best buffering-space performance among all the tested algorithms in practice, which is due to the fact that EQ eagerly evaluates queries and buffers only one physical copy of each potential answer node. In particular, compared to the other algorithms, the amount of buffering space EQ saves could be very large.

6. CONCLUSION

In this paper we revisited the XPath stream-querying problem and showed that a practical XPath fragment, Univariate XPath [3], can be efficiently evaluated in $O(|D||Q|)$ time in the streaming environment. Specifically, we proposed two stream-querying algorithms, LQ and EQ, which are based on the lazy strategy and on the eager strategy, respectively. To the best of our knowledge, our algorithms are the first XPath stream-querying algorithms that guarantee $O(|D||Q|)$ worst-case time performance. Moreover, LQ and EQ achieve $O(|D|\ |Q|)$ time performance without trading off space performance. Instead, they have better buffering-space performance than the state-of-the-art algorithms that use the lazy or eager strategy, respectively. In particular, EQ achieves the optimal buffering-space performance.

We showed that our algorithms are not only of theoretical value. The results of our extensive performance evaluation show that our algorithms have considerable practical time and space performance advantages over the state-of-the-art algorithms in presence of recursion. When there is no recursion, our algorithms' performance is guaranteed to be similar to that of the state-of-the-art algorithms. We have observed that our algorithms LQ and EQ show similar time performance in practice, but EQ has better buffering-space performance than LQ. Therefore, we select EQ as the best-performance representative of our algorithms. We are currently extending EQ to evaluate more expressive classes of XML queries over streams, such as multi-variate XPath [4], XPath with backward axes [5], and XQuery-like queries [22].

7. REFERENCES

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

[2] Apache. Apache Xerces2-Java XML Parser. *http://xerces.apache.org/xerces2-j/*.

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

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

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

[6] S. Boag, D. Chamberlin, M. F. Ferandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML Query Language. *W3C, http://www.w3.org/TR/xquery/*, 2003.

[7] D. Brownell and D. Megginson. SAX: Simple API for XML. *SAX Project Organization, http://www.saxproject.org/*.

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

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

[10] Y. Chen, S. B. Davidson, and Y. Zheng. An efficient XPath query processor for XML streams. *ICDE*, 2006.

[11] B. Choi. What are real DTDs like? *WebDB Workshop*, 2002.

[12] J. Clark. XML Transformations (XSLT) Version 1.0. *W3C, http://www.w3.org/TR/xslt/*, 1999.

[13] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. *W3C, http://www.w3.org/TR/xpath/*, 1999.

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

[15] A. L. Diaz and D. Lovell. IBM's XML generator. *http://www.alphaworks.ibm.com/tech/xmlgenerator*.

[16] M. Fernandez and et al. Galax: an implementation of XQuery. *http://www.galaxquery.org/*.

[17] D. Florescu, C. Hillery, D. Kossmann, P. Lucas, F. Riccardi, T. Westmann, M. J. Carey, A. Sundararajan, and G. Agrawal. The BEA/XQRL streaming XQuery processor. *VLDB Conference*, 2003.

[18] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. *VLDB Conference*, 2002.

[19] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. *Proceedings of PODS*, 2003.

[20] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems (TODS)*, 29:752–788, 2004.

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

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

[23] M. Kay. Saxon: the XSLT and XQuery processor. *http://saxon.sourceforge.net/*.

[24] 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 Conference*, 2004.

[25] B. Ludascher, P. Mukhopadhyay, and Y. Papakonstantinou. A transducer-based XML query processor. *VLDB*, 2002.

[26] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *http://www.cs.umd.edu/projects/xsq/*.

[27] F. Peng and S. S. Chawathe. XSQ: A streaming XPath engine. *ACM Transactions on Database Systems (TODS)*, 30:577–623, 2005.

[28] A. Schmidt and et al. XMark: an XML benchmark project. *http://monetdb.cwi.nl/xml/*.

[29] L. Segoufin. Typing and querying XML documents: some complexity bounds. *Proceedings of PODS*, 2003.

[30] D. Suciu. XML data repository. *http://www.cs.washington.edu/research/xmldatasets/*.

[31] W3C. Section 1.2.2, XML query use cases. *http://www.w3.org/TR/xquery-use-cases/*, 2006.