Buffering in Query Evaluation over XML Streams

Ziv Bar-Yossef
Dept. of Electrical Engineering
Technion
Haifa 32000, ISRAEL
zivby@ee.technion.ac.il

Marcus Fontoura
IBM Almaden
650 Harry Road
San Jose, CA 95120, USA.
fontoura@us.ibm.com

Vanja Josifovski
IBM Almaden
650 Harry Road
San Jose, CA 95120, USA.
vanja@us.ibm.com

ABSTRACT

All known algorithms for evaluating advanced XPath queries (e.g., ones with predicates or with closure axes) on XML streams employ buffers to temporarily store fragments of the document stream. In many cases, these buffers grow very large and constitute a major memory bottleneck. In this paper, we identify two broad classes of evaluation problems that independently necessitate the use of large memory buffers in evaluation of queries over XML streams: (1) full-fledged evaluation (as opposed to just filtering) of queries with predicates; (2) evaluation (whether full-fledged or filtering) of queries with "multi-variate" predicates.

We prove quantitative lower bounds on the amount of memory required in each of these scenarios. The bounds are stated in terms of novel document properties that we define. We show that these scenarios, in combination with query evaluation over recursive documents, cover the cases in which large buffers are required. Finally, we present algorithms that match the lower bounds for an important fragment of XPath.

1. INTRODUCTION

All known algorithms for evaluating XPath and XQuery queries over XML streams [2, 4, 9, 10, 14, 15, 18, 19, 20, 23, 25, 26] suffer from excessive memory usage on certain queries and documents. The bulk of memory used is dedicated to two tasks: (1) storage of large transition tables; and (2) buffering of document fragments. The former emanates from the standard methodology of evaluating queries by simulating finite-state automata. The latter is a result of the limitations of the data stream model.

Finite-state automata or transducers are the most natural mechanisms for evaluating XQuery/XPath queries. However, algorithms that explicitly compute the states of these automata and the corresponding transition tables incur memory costs that are exponential in the size of the query in the worst-case. The high costs are a result of the blowup in the transformation of non-deterministic automata into deterministic ones. In our previous paper [6] we investigated the space complexity of XPath evaluation on streams as a function of the query size, and showed that the exponential dependence is avoidable. We exhibited an optimal algorithm whose memory depends only linearly on the query size (for some types of queries, the dependence is even logarithmic).

In this paper we study the other major source of memory consumption: buffers of (representations of) document fragments. Algorithms that support advanced features of the XPath language, such as predicates, full-fledged evaluation (as opposed to only filtering), or closure axes, face the need to store fragments of the document stream during the evaluation. The buffering seems necessary, because in many cases at the time the algorithm encounters certain XML elements in the stream, it does not have enough information to conclude whether these elements should be part of the output or not (the decision depends on unresolved predicates, whose final value is to be determined by subsequent elements in the stream). Indeed, all the advanced evaluation algorithms maintain some form of buffers (e.g., the stack of the XPush Machine [15], the BPDT buffers of the XSQ system [26], the predicate buffers of TurboXPath [19], and the buffer trees of the FluX query engine [20]). It has been noted anecdotally [19, 26] that for certain queries and documents, buffering seems unavoidable. However, to date, there has not been any formal and theoretical study that quantifies the amount of buffering needed to support advanced features of XPath. That is the main focus of this paper.

Our contributions We identify two major classes of XPath evaluation problems that necessitate buffering. We prove space lower bounds that quantify the amount of buffering required in terms of new document properties that we introduce. Finally, we present an almost matching algorithm.

The two classes of evaluation problems are the following:

1. Full-fledged evaluation of queries with predicates: There are two typical modes of query evaluation: *full-fledged evaluation* (i.e., outputting all the document elements selected by the query) and *filtering* (i.e., outputting a bit indicating whether the query selects any node from the document or not). We prove that full-fledged evaluation of queries with predicates requires substantially more space than just filtering documents using such queries.

Our lower bound is stated in terms of a property of documents we call "concurrency". A document node is said to be "alive" after reading t events from the document stream, if these t events do not convey sufficient information to decide whether the node will be eventually selected by the query or not. For example, if the query is `/a[b]/c`, the 'c' children of an 'a' node in any document will be alive until a 'b' child is encountered or until the 'a' element is closed. The maximum number of nodes that are alive simultaneously is called the *concurrency* of the document w.r.t. the query. We prove that document concurrency is a lower bound on the space complexity of evaluating the query on XML streams.

We note that if a query has no predicates, the concurrency of any document w.r.t. that query is 1, while for a large class of queries with even a single predicate, it is possible to construct documents of concurrency that is linear in the document size. Therefore, our concurrency lower bound implies full-fledged evaluation of queries with predicates requires buffering, which could be up to linear in the document size.

In [6] we presented a filtering algorithm that works even for queries with ("univariate") predicates and that uses space that is only logarithmic in the document size. We thus conclude that the space required to fully evaluate a query could be exponentially higher than the space required to just filter documents using the same query.

2. Multi-variate comparison predicates: A predicate that consists of a comparison of two nodes (e.g., `a = b` or `a > b`) is said to be a *multi-variate comparison predicate*. On the other hand, *univariate comparison predicates* are ones that compare a node with a constant (e.g., `a = 5` or `b > 4`). We prove that evaluation (whether full-fledged or filtering) of queries that consist of multi-variate comparison predicates may require substantially more space than evaluation of queries that have only univariate comparison predicates.

The existential semantics of XPath implies that a predicate of the form `/c[`R`(a,b)]`, where R is any comparison operator (e.g., `=`,`>`), is satisfied if and only if the document has a 'c' node with at least one 'a' child with a value x and one 'b' child with a value y, so that $R(x, y) = \texttt{true}$. Thus, if all the 'a' children of the 'c' node precede its 'b' children, the evaluation algorithm may need to buffer the (distinct) values of the 'a' children, until reaching the first 'b' child. We prove that such a buffering is indeed necessary when R is an equality operator (i.e., `=`,`!=`). It is not needed for inequality operators (i.e., `<`,`<=`,`>`,`>=`), because for them it suffices to buffer just the maximum or minimum value of the 'a' children.

Our lower bound is in fact stated w.r.t. to any relational operator R, not just comparisons. The bound is given in terms of a graph-theoretic property of relations, which we call the "dominance cardinality".

The above lower bound implies that the space complexity of filtering documents using queries with multi-variate equality predicates can be up to linear in the document size. Our algorithm from the previous paper [6] shows that filtering documents using queries with only univariate predicates can be done with logarithmic space. Thus, multi-variate equality predicates may necessitate an exponential blowup in the space consumption of the filtering algorithm.

The two above classes of problems, together with query evaluation over recursive documents (as discussed in [6]), exhaust the cases in which buffering is required for XPath evaluation. The algorithm from the previous paper shows that in all other cases (i.e., filtering non-recursive documents using queries with only univariate predicates), no substantial buffering is necessary.

The techniques used to prove the lower bounds are based on reductions from the model of *one-way communication complexity*. Our first lower bound requires a twist on the model, in which the communicating parties output sets of elements on a write-only output tape, rather than a single element at the last message of the protocol.

Algorithm In order to demonstrate the tightness of our lower bounds, we present an XPath evaluation algorithm, which uses space close to the optimum for the classes of problems discussed above. The core idea of this algorithm is a sophisticated buffer management that allows eager evaluation of predicates. That is, the algorithm buffers document elements only as long as either: (1) it is not yet clear whether they will be selected by the query or not; or (2) their value may be required to evaluate pending predicates. The number of elements the algorithm thus buffers is proportional to the concurrency of the document, matching our lower bound.

Complexity measures Our concurrency lower bound is proved w.r.t. a particularly strong notion of complexity, which is reminiscent of the *instance optimality* of Fagin, Lotem, and Naor [11]. Ideally, we would like to show that for every query-document pair (Q, D), any evaluation algorithm would need to use $\textsc{concur}(D, Q)$ bits of space, when receiving (Q, D) as input (here, $\textsc{concur}(D, Q)$ is the concurrency of D w.r.t. Q). Such a statement though is impossible to prove. Consider the following algorithm $\mathcal{A}$: $\mathcal{A}$ has the query Q and the document D "hard-coded". It compares its input query to Q and its input stream to D, and as long as they match, it just continues to read the stream. Only if a mismatch is found, $\mathcal{A}$ starts to act like a normal evaluation algorithm. If the input is indeed (Q, D), $\mathcal{A}$ can output the elements Q selects from D, because it knows them a priori (they are also hard-coded in the algorithm). The space $\mathcal{A}$ uses on (Q, D) is constant, and in particular no buffers are used.

In order to get around such tricks, we prove an almost as powerful statement. We show that for every query Q and every document D, any algorithm that evaluates Q on XML streams uses $\textsc{concur}(D, Q)$ bits of space, when running on a document that is "almost-isomorphic" to D. D' is almost-isomorphic to D if it is the same as D, except for a few additional empty nodes, whose name does not appear as a node test in Q. One can show that Q cannot distinguish between almost-isomorphic documents. Furthermore, a document D may have many almost-isomorphic documents, and thus the "hard-coding" trick is not applicable any more.

Related work Two recent surveys [5, 24] provide extensive overview of the data stream model and its applications. Space lower bounds for this model are typically proved through the model of communication complexity (see, e.g., [1, 7, 16, 17, 27, 29]).

Several algorithms for evaluating XPath queries over document streams have been proposed [2, 4, 9, 10, 14, 15, 18, 25, 26]. None of these algorithms supports queries with multi-variate predicates, as our algorithm does. Recently, research on processing XPath over streams has expanded to XQuery [19, 20, 23]. Marian and Siméon [23] describe an XQuery evaluation algorithm that produces a succinct representation of the XML document in a pre-processing step, and then performs the evaluation on this representation. It is not clear how far this succinct representation is from the space lower bound. TurboXPath [19] evaluates XQuery queries over streams, but does not apply eager evaluation of predicates, and thus uses sub-optimal space in certain cases. The FluX system [20] uses the DTD schema of the document stream to reduce the size of buffers needed for the evaluation of XQuery queries. In our work, we focus on evaluation of queries on schema-less input documents.

The space complexity of XPath evaluation over ordinary (i.e., non-streaming) XML documents has been studied by Gottlob, Koch, and Pichler [13] and by Segoufin [28]. The only previous work to provide space lower bounds for XPath evaluation over streams is our own previous work [6], which as explained above focused on the dependence of the memory usage on the query size. None of the approaches used in these papers is applicable to buffering lower bounds.

Arasu *et al.* [3] studied memory lower bounds for evaluation of continuous queries over multiple data streams. While the settings of the two projects are different, (select-project-join vs. XPath queries) some of the challenges are similar. In particular, also in their context multi-variate predicates necessitate large buffers. We note, however, that their goals were much more coarse-grained: distinguishing between queries that require constant ("bounded") space versus ones that require linear space. We, on the other hand, give exact quantifications of the memory required.

The rest of the paper is organized as follows. In Section 2 we review necessary preliminaries. In Section 3 we prove the concurrency lower bound. In Section 4 we prove the "dominance cardinality" lower bound. We present the algorithm in Section 5, and end with concluding remarks in Section 6. For lack of space, a few of the proofs are omitted and are to appear in the full version of this paper.

2. PRELIMINARIES

Notation Queries and documents are modeled as rooted trees. We will use the letters u, v, w to denote query nodes and the letters x, y, z to denote document nodes. For a tree T, $\text{ROOT}(T)$ is the root of T. For a node $x \in T$, $\text{PATH}(x)$ is the sequence of nodes on the path from the root to x.

$\mathcal{N}$ is the set of all legal XML node names, $\mathcal{S}$ is the set of all finite-length strings of UCS characters, and $\mathcal{V}$ is the set of atomic data values (numbers, strings, booleans, etc.) that XML supports.

For integers $i \le j$, $[i..j]$ denotes the set $\{i, \ldots, j\}$. For a function $f : A \to B$, and for a subset $S \subseteq A$ of the domain, $f(S) \overset{\text{def}}{=} \{f(a) \mid a \in S\}$.

XML We use the XPath 2.0 and XQuery 1.0 Data Model [12]. An XML document is a rooted tree. Every node x has the following properties: (1) $\text{KIND}(x)$, which in this paper can be either `root`, `element`, `attribute`, or `text`. The root and only the root is of kind `root`. `text` and `attribute` nodes are always leaves and are associated with *text contents*, which are strings from $\mathcal{S}$. (2) $\text{NAME}(x)$, which is a value from $\mathcal{N}$. `root` and `text` nodes are unnamed. (3) $\text{STRVAL}(x)$, which is a string from $\mathcal{S}$. $\text{STRVAL}(x)$ is the concatenation of the text contents of the text node descendants of x in "document order" (i.e., pre-order traversal). (4) $\text{DATAVAL}(x)$, which is a data value from $\mathcal{V}$. $\text{DATAVAL}(x)$ is derived from $\text{STRVAL}(x)$, using the document's XML schema.

XPath Figure 1 describes *Forward XPath*—a fragment of XPath 2.0 [8], which supports only the forward axes. All the XPath fragments considered in this paper are subsets of Forward XPath.

```
Path         :=  Step | Path Step
RelPath      :=  RelStep | RelPath Step
Step         :=  Axis NodeTest ('[' Predicate ']')?
RelStep      :=  RelAxis NodeTest ('[' Predicate ']')?
Axis         :=  '/' | '//' | '@'
RelAxis      :=  './/' | '@'
NodeTest     :=  name | '*'
Predicate    :=  Expression |
                 Expression compop Expression |
                 Predicate 'and' Predicate |
                 Predicate 'or' Predicate |
                 'not(' Predicate ')'
Expression   :=  const | RelPath |
                 Expression arithop Expression |
                 '-' Expression |
                 funcop '(' Expression?
                        (',' Expression)* ')'
```

`name` is any string from $\mathcal{N}$.
`const` is any string from $\mathcal{S}$.
`compop` ∈ { `=, !=, <, <=, >, >=` }.
`arithop` ∈ { `+, -, *, div, idiv, mod` }.
`funcop` is any basic XPath function or operator on atomic arguments as specified in [22], excluding the functions `position()` and `last()`.

Figure 1: Grammar of Forward XPath.

An XPath query is a rooted tree. Each node u has the following properties: (1) $\text{AXIS}(u)$, which in this paper can be either `child`, `attribute`, or `descendant`.[1] The root does not have an axis. (For the remainder of the paper, we omit explicit treatment of the `attribute` axis, because it can be handled as a special case of the `child` axis.) (2) $\text{NTEST}(u)$, which is either a name from $\mathcal{N}$ or the wildcard `*`. The root does not have a node test. (3) $\text{SUCCESSOR}(u)$, which is either empty or one of the children of u. (4) $\text{PREDICATE}(u)$, which is either empty or an expression tree, as described below.

$\text{PREDICATE}(u)$ is an expression tree whose internal nodes are labeled by logical, comparison, arithmetic, or functional operators, and whose leaves are labeled by constants from $\mathcal{V}$ or by pointers to children of u. The XPath semantics requires that all the children of u, except for the successor, are pointed to by leaves of the predicate. They are called the *predicate children* of u. No two leaves of the predicate can point to the same child of u.

The arguments and the output of every operator are associated with types. These types can be either *atomic* (e.g., numbers, strings, booleans) or *sequences* (sequences of atomic values).

The successor-less node reached by repeatedly following successors from a given node u is called the *succession leaf of* u, and is denoted by $\text{LEAF}(u)$. The succession leaf of the root is called the *query output node*, and is denoted by $\text{OUT}(Q)$. Nodes that are not successors of their parents are called *succession roots*. A node is a succession root, if it is either the root of the query or a predicate child of its parent. The sequence of nodes from a succession root u to

[1]Our results can be extended to also handle the `self` and `descendant-or-self` axes. We chose not to do that, in order to keep the presentation more clean and clear.

the succession leaf $\textsc{leaf}(u)$ is called a *succession path*. For example, in the query `/a[b/c > 5 and d[e] > 7]/f` there are four succession paths: (1) `a`, `f`; (2) `b`, `c`; (3) `d`; and (4) `e`.

Query evaluation An *evaluation* of a query Q on a document D is the sequence of nodes that Q "selects" from D. In this paper we use the notion of "matchings" in order to define this sequence.[2]

Definition 1. A *matching* of a document node x with a query node u is a mapping ϕ from $\textsc{path}(u)$ to $\textsc{path}(x)$ (recall our conventional notations) with the following properties:

1. **Root match:** $\phi(\textsc{root}(Q)) = \textsc{root}(D)$.
2. **Node test passage:** For all $v \in \textsc{path}(u)$ (except for the root), if $\textsc{ntest}(v) \neq *$, then $\textsc{name}(\phi(v)) = \textsc{ntest}(v)$.
3. **Axis match:** For all $v \in \textsc{path}(u)$ (except for the root), $\phi(v)$ relates to $\phi(\textsc{parent}(v))$ along $\textsc{axis}(v)$. That is, if $\textsc{axis}(v) = \texttt{child}$ (resp., $\textsc{axis}(v) = \texttt{descendant}$), then $\phi(v)$ is a child (resp., descendant) of $\phi(\textsc{parent}(v))$.
4. **Predicate satisfaction:** For all $v \in \textsc{path}(u)$,
$$\texttt{EBV}(\textsc{peval}(r_v, \phi(v))) = \texttt{true},$$
where r_v is the root of $\textsc{predicate}(v)$. Here, $\texttt{EBV}(\cdot)$ is the *effective boolean value* function, which converts arbitrary values into boolean, and $\textsc{peval}(\cdot,\cdot)$ is the *predicate evaluation function*, which evaluates predicates over document nodes. For exact specification, see the XPath reference [8] or the full version of our previous paper [6].
5. **Target match:** $\phi(u) = x$.

A mapping ϕ that satisfies all the above conditions, except for predicate satisfaction, is called a *structural matching.*

A query node may have multiple possible matchings with document nodes. In Figure 2 the query node named `b`' can be matched with any of the two '`b`' nodes in the document that satisfy the predicate.

Definition 2. The *evaluation of a query Q on a document D*, denoted $\textsc{eval}(Q, D)$, is the sequence of all document nodes x, for which there exists a matching with $\textsc{out}(Q)$. The nodes are ordered in document order.

XML streams A streaming algorithm for evaluating XPath on XML documents accepts its input document as a stream of *SAX events.* The algorithm can read the events only in the order they come, and cannot go backwards on the stream. Thus, the only way to remember previously seen events is to store them in memory. The algorithm has random access to the query. There are five types of SAX events: (1) `startDocument()`. (2) `endDocument()`. (3) `startElement(n)`, where $\texttt{n} \in \mathcal{N}$ (also denoted $\langle n \rangle$). (4) `endElement(n)` (also denoted $\langle /n \rangle$). (5) $\texttt{text}(\alpha)$, where $\alpha \in \mathcal{S}$ (also denoted α).

[2]This view only slightly deviates from the standard XPath specifications. Discussion of the relationship between the two models is deferred to the full version of the paper. For a similar discussion, see the full version of our previous paper [6]. We use this view for its simplicity. We believe that using the standard model would not have made a significant difference w.r.t. the results of this paper.

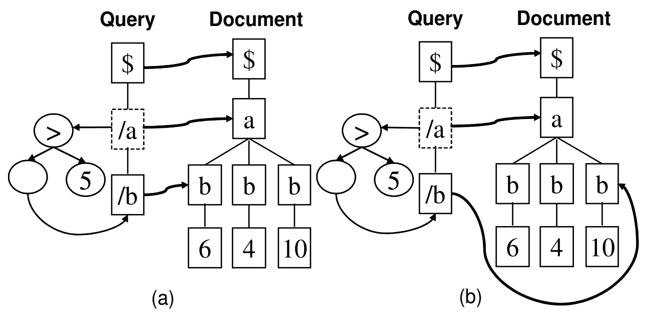


Figure 2: Two matchings of a the 'b' node in the query `/a[b > 5]`

On query Q and document D, the evaluation algorithm is supposed to output the following: $\{\textsc{ref}(x) \mid x \in \textsc{eval}(Q, D)\}$. Three remarks are in order. (1) $\textsc{ref}(x)$ should be, in principle, a reference to the node x in a Query Data Model (QDM) representation of D [12]. However, when evaluating queries over streams it is typically impractical to store the whole stream as a QDM instance. Thus, current streaming implementations output, for each selected node x, either its full content, its unique node identity (if available), or its string value. In this paper, we will assume the streaming algorithms output the contents of selected nodes. Everything we do can be extended also to the other cases. (2) We do not require the algorithm to output the selected nodes in "document order", but rather in arbitrary order. This makes our lower bounds even stronger. (3) We require the algorithm to eliminate duplicates: if a node x is selected via several matchings, the algorithm has to output it only once.

Communication complexity In the communication complexity model [21, 30] two players, Alice and Bob, jointly compute a function $f : \mathcal{A} \times \mathcal{B} \to \mathcal{Z} \cup \{\bot\}$. Alice is given $\alpha \in \mathcal{A}$ and Bob is given $\beta \in \mathcal{B}$, and they exchange messages according to a protocol. If $f(\alpha, \beta) \neq \bot$, then (α, β) is called a "well-formed" input, and then the last message sent in the protocol should be the value $f(\alpha, \beta)$. Otherwise, the last message can be arbitrary. The cost of the protocol is the maximum number of bits (over all (α, β)) Alice and Bob send to each other. The *communication complexity* of f, denoted $CC(f)$, is the minimum cost of a protocol that computes f.

A *one-way communication protocol* is one in which Alice sends a single message to Bob, and then Bob sends a single message to Alice consisting of the output $f(\alpha, \beta)$. The *one-way communication complexity* of f is denoted by $CC_1(f)$.

In this paper we discuss *set-computing communication protocols.* Such protocols fit problems, in which the output $f(\alpha, \beta)$ is a set of elements from a universe U. In these protocols, the output is not given in the last message, but rather on a special write-only output tape. Throughout the execution of the protocol, each of the two players writes elements of U to the output tape, but cannot read the elements the other player has written. If the set Alice outputs is S_A and the set Bob outputs is S_B, the requirement is that $S_A \cup S_B = f(\alpha, \beta)$. Alice and Bob are allowed to output the same element several times.

3. CONCURRENCY LOWER BOUND

In this section, we quantify the amount of space needed to perform full-fledged evaluation of XPath queries in terms of the "document concurrency". The lower bound applies to evaluation of queries in *Star-Free XPath*, which is the subset of Forward XPath consisting of queries that do not have wildcard node tests. Throughout the section, we fix Q to be an arbitrary query in this fragment of XPath.

For a given sequence of SAX events $\overline{\alpha} = (\alpha_1, \ldots, \alpha_m)$, we denote by $\textsc{doc}(\overline{\alpha})$ the document (or document fragment) represented by this sequence. Conversely, for a given document (or document fragment) D, we denote by $\textsc{stream}(D)$ the sequence of SAX events representing D. For two event sequences, $\overline{\alpha}$ and $\overline{\beta}$, we denote by $\overline{\alpha} \circ \overline{\beta}$ the sequence obtained by concatenating $\overline{\alpha}$ and $\overline{\beta}$.

Fix a document D, and let $(\alpha_1, \ldots, \alpha_m) = \textsc{stream}(D)$. We view the m indices $1, \ldots, m$ as "time steps". For any node $x \in D$ of kind `root` or `element`, we denote by $\textsc{beg}(x)$ and $\textsc{end}(x)$ the time steps of the events corresponding, respectively, to its `startElement` and `endElement` events. The document segment starting at step $\textsc{beg}(x)$ and ending at step $\textsc{end}(x)$ is called the *content* of x, and is denoted by $\textsc{cont}(x)$. Two nodes x, x' are said to be *content-distinct*, if $\textsc{cont}(x) \neq \textsc{cont}(x')$.

Definition 3. Suppose $x \in D$ structurally matches $\textsc{out}(Q)$. Let $t \geq \textsc{end}(x)$, and let $\overline{\alpha} = (\alpha_1, \ldots, \alpha_t)$ be the first t events in $\textsc{stream}(D)$. Note that x belongs to any document that is represented by an event stream whose prefix is $\overline{\alpha}$.

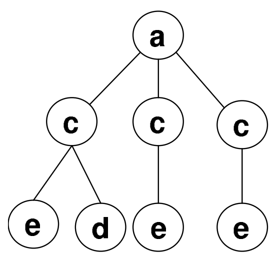
We say that x *is alive at step* t, if: (1) there exists an event sequence $\overline{\beta}$, so that $\textsc{doc}(\overline{\alpha} \circ \overline{\beta})$ is a well-formed document and $x \in \textsc{eval}(Q, \textsc{doc}(\overline{\alpha} \circ \overline{\beta}))$; and (2) there exists an event sequence $\overline{\gamma}$, so that $\textsc{doc}(\overline{\alpha} \circ \overline{\gamma})$ is a well-formed document and $x \notin \textsc{eval}(Q, \textsc{doc}(\overline{\alpha} \circ \overline{\gamma}))$.

Definition 4. The *concurrency* of the document D w.r.t. to Q at step $t \in [1..m]$, is the number of content-distinct nodes in D that are alive at step t. The *concurrency* of D w.r.t. Q, denoted $\textsc{concur}(D, Q)$, is the maximum concurrency, over all steps $t \in [1..m]$.

To illustrate the above definitions, consider the following example. Let Q = `/a[b]/c[d]/e` and let D be the document depicted in Figure 3. At time step 14 (when the last 'e' element ends), there are two live elements: the first 'e' element (which starts on line 3) is alive, because whether it will be selected depends on whether its 'a' grandparent will have a 'b' child. The third 'e' element (which starts on line 13) is alive, because whether it will be selected depends on whether its 'c' parent will have a 'd' child and its 'a' grandparent will have a 'b' child. The second 'e' element (which starts on line 9) is not alive at step 14, because it is not going to be selected by the query, irrespective of the rest of the document (its 'c' parent does not have a 'd' child). The concurrency of this document w.r.t. Q at step 14 is therefore 2, and this is also its maximum concurrency.

Remark 1. It is easy to verify that if Q has no predicates, then the concurrency of any document w.r.t. Q is 1. On the other hand, for many queries Q that have even a single predicate (e.g., Q = `/a[b]/c`), it is possible to construct documents that have arbitrarily large concurrency.

A document D is called *recursive*, if it has two nodes with the same name that are nested one within the other.



1:	⟨a⟩	9:	⟨e⟩
2:	⟨c⟩	10:	⟨/e⟩
3:	⟨e⟩	11:	⟨/c⟩
4:	⟨/e⟩	12:	⟨c⟩
5:	⟨d⟩	13:	⟨e⟩
6:	⟨/d⟩	14:	⟨/e⟩
7:	⟨/c⟩	15:	⟨/c⟩
8:	⟨c⟩	16:	⟨/a⟩

Figure 3: Example Document

Our lower bound shows that for any query Q and any non-recursive document D, any algorithm that evaluates Q on XML streams will need to use $\Omega(\textsc{concur}(D, Q))$ bits of space when running on D or on a "closely related" document. We next formally define this notion of "closeness".

Definition 5. A document D' is called *almost-isomorphic* to D w.r.t. Q, if it is identical to D, except for extra "benign nodes". A node is called *benign*, if: (1) it is of kind `element`; (2) it is empty; and (3) its name does not appear as a name of any node in D or as a node test of any node in Q.

If D' is almost-isomorphic to D, then there is a 1-1 embedding π of D into D'. The following lemma, whose proof will appear in the full version of this paper, shows that the query Q cannot distinguish between almost-isomorphic documents:

Lemma 1. *If a document D' is almost-isomorphic to a document D w.r.t. Q, then* $\textsc{eval}(Q, D') = \pi(\textsc{eval}(Q, D))$.

We are now ready to state our main theorem:

Theorem 1 (Main theorem).
For every Star Free XPath query Q, for every non-recursive document D, and for any algorithm $\mathcal{A}$ that evaluates Q on XML streams, there is a document D', which is almost-isomorphic to D w.r.t. Q, on which $\mathcal{A}$ uses $\Omega(\textsc{concur}(D, Q))$ bits of space.

Proof. The lower bound will be proved by a reduction from the following set-computing communication complexity problem. In the problem $\mathcal{P}$, Alice and Bob compute a two-argument function $p(\cdot, \cdot)$, defined as follows. Alice's input is a subset $S \subseteq [1..C]$, Bob's input is a bit $b \in \{0, 1\}$, and $p(S, b)$ is defined to be S, if $b = 1$, and $\emptyset$ otherwise. We first prove that the one-way communication complexity of $\mathcal{P}$ is at least C:

Proposition 1. $CC_1(\mathcal{P}) \geq C$.

Proof. Let Π be any protocol for $\mathcal{P}$. Let $m_\Pi(S)$ denote the message Alice sends on input S in this protocol, let $A_\Pi(S)$ denote the output Alice generates on input S in this protocol, and let $B_\Pi(m, b)$ denote the output Bob generates when receiving a message m from Alice and an input b in this protocol. Since Π correctly computes $\mathcal{P}$, then for any pair of inputs (S, b), the output of the protocol, $A_\Pi(S) \cup B_\Pi(m_\Pi(S), b)$, should equal $p(S, b)$. The cost of the protocol is $\max_S |m_\Pi(S)|$.

We would like to prove that there are no two inputs S, S' of Alice, for which $m_\Pi(S) = m_\Pi(S')$. It would then immediately follow that the communication complexity of $\mathcal{P}$

is at least C. Suppose, to reach a contradiction, that there are two inputs $S \neq S'$ for which $m_\Pi(S) = m_\Pi(S')$. Since $p(S, 0) = p(S', 0) = \emptyset$, then $A_\Pi(S) = A_\Pi(S') = \emptyset$ as well. Thus, Bob is the only one who can generate output in this protocol. Now, since Bob sees the same message from Alice on the inputs S, S', he will generate the same output: $B_\Pi(m_\Pi(S), 1) = B_\Pi(m_\Pi(S'), 1)$. Hence, $p(S, 1) = p(S', 1)$. However, $p(S, 1) = S$ and $p(S', 1) = S'$, and $S \neq S'$. $\square$

Next, we describe the reduction. Let $C = \textsc{concur}(D, Q)$, let t be the step at which D has the maximum concurrency w.r.t. Q, let L be the set of nodes that are alive at step t, and let $R = \{r_1, \ldots, r_C\}$ be the C distinct contents of these nodes. For each $i \in [1..C]$, let L_i be the nodes in L whose content is r_i. For each subset $S \subseteq [1..C]$, let $L_S = \cup_{i \in S} L_i$. We define a document D_S to be the same as D, except for the nodes in L_S, to each of which we add an empty `element` child whose name we denote by '\$' ('\$' denotes a name in $\mathcal{N}$ which will be determined later; this name does not appear as a name of any node in D or as a node test of any node in Q). We will show a reduction from the communication problem $\mathcal{P}$ to the evaluation of Q on the documents $\{D_S\}_S$. Note that these documents are all almost-isomorphic to D.

In order to carry out the reduction we need to show that an algorithm that evaluates Q on D_S can be used as a *black box* to "reveal" the set S. This will immediately translate into a communication protocol for $\mathcal{P}$. Since only black box use of the evaluation algorithm is allowed, we need to make sure that the set S can be extracted from the output of the algorithm. The output of the algorithm consists of the contents of nodes that are selected by the query. Thus, if all the nodes in L are selected by the query, then those of the nodes that are "marked" by the additional benign child will indicate the set S. The technical challenges we need to address are: (1) how to make sure that all the nodes in L are selected by Q? (2) prove that the addition of benign children does not affect the identity of the nodes selected by Q.

Let $(\alpha_1, \ldots, \alpha_m) = \textsc{stream}(D)$, and let $\overline{\alpha} = (\alpha_1, \ldots, \alpha_t)$ be the first t events in $\textsc{stream}(D)$. By definition, the endElement events corresponding to the nodes that are alive at step t belong to $\overline{\alpha}$. Thus, the set L is contained in any document whose prefix is $\overline{\alpha}$. The following lemma, whose proof appears below, addresses our first challenge:

Lemma 2. *There exist event sequences $\overline{\delta}_0$ and $\overline{\delta}_1$, such that:*

1. *$\forall b$, the document $E_b \stackrel{\text{def}}{=} \textsc{doc}(\overline{\alpha} \circ \overline{\delta}_b)$ is a well-formed document.*
2. *$L \subseteq \textsc{eval}(Q, E_1)$.*
3. *$L \cap \textsc{eval}(Q, E_0) = \emptyset$.*

The name '\$' will be chosen to be any name that does not appear either as: (1) a name of a node in D, E_0, or E_1; (2) a node test of a node in Q.

Recall that for each $i \in [1..C]$, r_i is the content of nodes in L_i. Let $\hat{r}_i$ denote the content of these nodes after the addition of the '\$' child. Also, let $\hat{R} = \{\hat{r}_1, \ldots, \hat{r}_C\}$ and let $\hat{R}_S = \{\hat{r}_i \mid i \in S\}$. The following lemma, whose proof appears below, addresses our second challenge:

Lemma 3. *There exist sequences $\overline{\gamma}_S$, for $S \subseteq [1..C]$, such that:*

1. *$\forall S$, $\overline{\gamma}_S$ is a prefix of $\textsc{stream}(D_S)$.*
2. *$\forall S, \forall b$, the document $E_{S,b} \stackrel{\text{def}}{=} \textsc{doc}(\overline{\gamma}_S \circ \overline{\delta}_b)$ is a well-formed document and is almost-isomorphic to E_b.*
3. *$\forall S$, $\textsc{cont}(\textsc{eval}(Q, \textsc{doc}(\overline{\gamma}_S \circ \overline{\delta}_1))) \cap \hat{R} = \hat{R}_S$.*
4. *$\forall S$, $\textsc{cont}(\textsc{eval}(Q, \textsc{doc}(\overline{\gamma}_S \circ \overline{\delta}_0))) \cap \hat{R} = \emptyset$.*

Let us show how to use Lemma 3 for the reduction. Suppose the algorithm $\mathcal{A}$ uses at most k bits of space to evaluate Q on each of the documents $\{D_S\}_S$. This means that k bits are enough to represent the state of $\mathcal{A}$ at any point of its execution on these documents. We next use $\mathcal{A}$ to construct a communication protocol Π for $\mathcal{P}$ whose cost is at most k. By Proposition 1 above, that would imply that $k \geq C$.

In the protocol Π, given the set S, Alice generates the sequence $\overline{\gamma}_S$, and runs $\mathcal{A}$ on it. After all the events in $\overline{\gamma}_S$ have been read by $\mathcal{A}$, Alice sends the state of $\mathcal{A}$ to Bob. Bob, upon receiving his input bit b, generates the sequence $\overline{\delta}_b$. When he gets the state of $\mathcal{A}$ from Alice, he resumes the execution of $\mathcal{A}$ and runs it on $\overline{\delta}_b$. During the execution of $\mathcal{A}$, if $\mathcal{A}$ outputs the content of a a node that has a benign child, then this content must equal $\hat{r}_i$ for some $i \in [1..C]$. Thus, whenever such a node is output by $\mathcal{A}$, the executing party (Alice or Bob) outputs i.

Note that the output of the protocol on (S, b) is

$$\textsc{cont}(\textsc{eval}(Q, \textsc{doc}(\overline{\gamma}_S \circ \overline{\delta}_b))) \cap \hat{R}.$$

By Conditions (3) and (4) of Lemma 3, this immediately implies that Π correctly computes p. At most k bits are needed to represent the state of $\mathcal{A}$ after processing $\overline{\gamma}_S$, because $\overline{\gamma}_S$ is a prefix of $\textsc{stream}(D_S)$ (Condition (1)). Thus, Π indeed uses at most k bits of communication. The reduction follows. $\square$

Proof of Lemma 2. Let x_{FIRST} and x_{LAST} be, respectively, the first and the last nodes in L in "document order". Since x_{FIRST} and x_{LAST} are alive at step t, there exist event sequences, $\overline{\beta}$ and $\overline{\gamma}$, so that: $x_{\text{FIRST}} \notin \textsc{eval}(Q, \textsc{doc}(\overline{\alpha} \circ \overline{\beta}))$ and $x_{\text{LAST}} \in \textsc{eval}(Q, \textsc{doc}(\overline{\alpha} \circ \overline{\gamma}))$. We define: $\overline{\delta}_0 = \overline{\beta}$ and $\overline{\delta}_1 = \overline{\gamma}$.

By definition, the documents $E_0 = \textsc{doc}(\overline{\alpha} \circ \overline{\delta}_0)$ and $E_1 = \textsc{doc}(\overline{\alpha} \circ \overline{\delta}_1)$ are well-formed. We are left to prove that $L \subseteq \textsc{eval}(Q, E_1)$ and $L \cap \textsc{eval}(Q, E_0) = \emptyset$. To this end, we prove the following:

Claim 1. *Let x and x' be two nodes in L, and suppose x precedes x' in document order. Let $\overline{\beta}$ be any event sequence that complements $\overline{\alpha}$ into a well formed document $E = \textsc{doc}(\overline{\alpha} \circ \overline{\beta})$. If $x' \in \textsc{eval}(Q, E)$, then also $x \in \textsc{eval}(Q, E)$.*

Before we prove the claim, let us use it to complete the proof of the lemma. Since $x_{\text{LAST}} \in \textsc{eval}(Q, E_1)$, and since all the other nodes in L precede x_{LAST} in document order, then by the above claim all of them must belong to $\textsc{eval}(Q, E_1)$. On the other hand, since $x_{\text{FIRST}} \notin \textsc{eval}(Q, E_0)$, and since x_{FIRST} precedes all other nodes in L, then by the above claim none of them belongs to $\textsc{eval}(Q, E_0)$. $\square$

Proof of Claim 1. Let $\overline{\beta}$ be any event sequence that complements $\overline{\alpha}$ into a well-formed document, and let $E = \textsc{doc}(\overline{\alpha} \circ \overline{\beta})$. Recall that the subtrees rooted at nodes in L are fully contained in the document segment represented by $\overline{\alpha}$, and therefore all of them are contained in E. Furthermore, the `startElement` events of all the ancestors of nodes in L belong to $\overline{\alpha}$, and thus these ancestors also belong to E (though, the subtrees rooted at these ancestors may be different from the corresponding subtrees in D).

Recall also that all the nodes in L structurally match $\text{OUT}(Q)$ in the document D, and thus each node $x \in L$ has a structural matching ϕ_x with $\text{OUT}(Q)$. Since x and all its ancestors belong both to D and to E, ϕ_x is also a structural matching of x and $\text{OUT}(Q)$ in E. We start by proving that ϕ_x is unique:

CLAIM 2. *For all nodes $x \in L$, there is a unique structural matching ϕ_x of x and* $\text{OUT}(Q)$.

PROOF. Suppose, to reach a contradiction, there exists two structural matchings ϕ_x and ϕ'_x. Let $\text{PATH}(x) = x_1, \ldots, x_\ell$ be the path from $\text{ROOT}(E)$ to x. Let $\text{PATH}(\text{OUT}(Q)) = u_1, \ldots, u_k$ be the path from $\text{ROOT}(Q)$ to $\text{OUT}(Q)$ (i.e., $u_1 = \text{ROOT}(Q)$ and $u_k = \text{OUT}(Q)$). Since $\phi_x \neq \phi'_x$, there is an index j so that $\phi_x(u_j) \neq \phi'_x(u_j)$. Suppose, for example, that $\phi_x(u_j) = x_r$, $\phi'_x(u_j) = x_{r'}$, and $r < r'$. x_r must then be an ancestor of $x_{r'}$. Furthermore, x_r and $x_{r'}$ must have the same name, because they both pass the node test of u_j, and this node test is not the wildcard (recall that Q is star-free). This means that the documents D and E are recursive, in contradiction to our assumption that D is non-recursive. □

Recall that x and x' are two nodes in L and x precedes x' in document order. Let y be the lowest common ancestor of x and x', and let $y_1, \ldots, y_\ell = \text{PATH}(y)$ be the path from $\text{ROOT}(E)$ to y. Let $u_1, \ldots, u_k = \text{PATH}(\text{OUT}(Q))$, and let j_x be the index of the last node among $u_1, \ldots, u_k$ to be mapped by the structural matching ϕ_x to a node in $\text{PATH}(y)$ (j_x is well-defined because $\phi_x(u_1) = \text{ROOT}(E)$ always belongs to $\text{PATH}(y)$). We similarly define $j_{x'}$. The following claim shows that the two structural matchings ϕ_x and $\phi_{x'}$ must agree on the nodes that map to $\text{PATH}(y)$:

CLAIM 3. *$j_x = j_{x'}$ and $\phi_x, \phi_{x'}$ agree on $u_1, \ldots, u_j$ (where $j = j_x = j_{x'}$).*

PROOF. Suppose, to reach a contradiction, the statement does not hold. Let r be an index so that $\phi_x(u_r) \neq \phi_{x'}(u_r)$ and at least one of the two belongs to $\text{PATH}(y)$. Suppose, for example, that $\phi_x(u_r) \in \text{PATH}(y)$. This means that $\phi_x(u_r)$ equals to y or is an ancestor of y. $\phi_{x'}(u_r)$ belongs to $\text{PATH}(x')$—the path from the root to x'. Since y is an ancestor of x', $\phi_{x'}(u_r)$ must lie on the same root-to-leaf path as y. Since $\phi_x(u_r)$ is ancestor-or-self of y, $\phi_{x'}(u_r)$ and $\phi_x(u_r)$ lie on the same root-to-leaf path. However, these two nodes are distinct and share the same name (because both pass the node test of u_r, which is not a wildcard). Therefore, the documents D and E are recursive, in contradiction to our assumption that D is non-recursive. □

Let $j = j_x = j_{x'}$. We know by the above claim that both ϕ_x and $\phi_{x'}$ map $u_1, \ldots, u_j$ to $\text{PATH}(y)$. We next prove that the matching ϕ_x must satisfy all the predicates of $u_{j+1}, \ldots, u_k$:

CLAIM 4. *For every $\ell \in [j+1..k]$, the node $\phi_x(u_\ell)$ satisfies* $\text{PREDICATE}(u_\ell)$.

PROOF. Fix any $\ell \in [j+1..k]$. Our XPath fragment allows only forward axis, and thus whether $\phi_x(u_\ell)$ satisfies $\text{PREDICATE}(u_\ell)$ depends only on nodes that are in the subtree rooted at $\phi_x(u_\ell)$. Since $\phi_x(u_\ell)$ is an ancestor of x and not an ancestor of x', and since x precedes x' in document order, this subtree is fully contained in the prefix $\overline{\alpha}$. Thus, whether $\phi_x(u_\ell)$ satisfies $\text{PREDICATE}(u_\ell)$ depends only on $\overline{\alpha}$ and is independent of the suffix $\overline{\beta}$. Now, if $\phi_x(u_\ell)$ does not satisfy $\text{PREDICATE}(u_\ell)$ in E, it means that also in any continuation $\overline{\beta}'$ of $\overline{\alpha}$ into a well-formed document, $\phi_x(u_\ell)$ does not satisfy $\text{PREDICATE}(u_\ell)$ in $\text{DOC}(\overline{\alpha} \circ \overline{\beta}')$. By Claim 2, we know that ϕ_x is the only matching x can have with $\text{OUT}(Q)$ in any document whose prefix is $\overline{\alpha}$. We conclude that if $\phi_x(u_\ell)$ does not satisfy $\text{PREDICATE}(u_\ell)$ in E, then x cannot be selected by Q on any document whose prefix is $\overline{\alpha}$. This is a contradiction to the assumption that x is alive at step t. □

Recall that y is the lowest common ancestor of the nodes x and x'. By Claim 3, there is a $j \in [1..k]$, so that both ϕ_x and $\phi_{x'}$ map $u_1, \ldots, u_j$ to $\text{PATH}(y)$. Since $\phi_{x'}$ is the only possible matching of x' and $\text{OUT}(Q)$ (Claim 2), and since x' is selected by Q on E, then $\phi_{x'}$ must satisfy the predicates of all the nodes $u_1, \ldots, u_k$. Since ϕ_x and $\phi_{x'}$ agree on $u_1, \ldots, u_j$ (Claim 3), then also ϕ_x satisfies the predicates of $u_1, \ldots, u_j$. By Claim 4, ϕ_x also satisfies the predicates of $u_{j+1}, \ldots, u_k$. We conclude that ϕ_x satisfies the predicates of all nodes $u_1, \ldots, u_k$, and thus x is selected by Q on E.

PROOF OF LEMMA 3. We start by describing the construction of the sequences $\{\overline{\gamma}_S\}_S$, and then prove they satisfy the four conditions.

Let $m_S = |\text{STREAM}(D_S)|$. Since D_S is the same as D plus some extra nodes, $\text{STREAM}(D)$ is a sub-sequence of $\text{STREAM}(D_S)$. Let π_S be the 1-1 mapping from $[1..m]$ to $[1..m_S]$ which specifies this sub-sequence. $\overline{\gamma}_S$ is defined to be the first $\pi_S(t)$ events in $\text{STREAM}(D_S)$.

We now prove that the four conditions are met.

(1) $\overline{\gamma}_S$ is a prefix of $\text{STREAM}(D_S)$ by definition.

(2) $\overline{\gamma}_S$ is identical to the event sequence $\overline{\alpha}$, except for the addition of empty '\$' children to nodes in L. These additions are self-contained (i.e., there are no dangling `startElement` events in $\overline{\alpha}$). Therefore, any continuation $\overline{\beta}$ of $\overline{\alpha}$ into a well-formed document also complements $\overline{\gamma}_S$ into a well-formed document. In particular, the documents $E_{S,0}$ and $E_{S,1}$ are well-formed.

The only difference between $E_{S,b}$ and E_b is the addition of empty `element` nodes whose name is '\$'. Since this string does not occur as a name of any node in E_b or as a node test of any node in Q, $E_{S,b}$ is almost-isomorphic to E_b w.r.t. Q.

(3) Let $\pi_{S,1}$ be the embedding of E_1 into $E_{S,1}$. By Lemma 1, we know that $\text{EVAL}(Q, E_{S,1}) = \pi_{S,1}(\text{EVAL}(Q, E_1))$. Hence, from Lemma 2, we have that $\pi_{S,1}(L) \subseteq \text{EVAL}(Q, E_{S,1})$. Now, $\pi_{S,1}(L_S)$ is the set of nodes of $E_{S,1}$ that have a '\$' child. Since all of them belong to $\text{EVAL}(Q, E_{S,1})$, then the set of content strings of nodes in $\text{EVAL}(Q, E_{S,1})$, which have a '\$' child, is exactly the set of content strings of nodes in $\pi_{S,1}(L_S)$. The latter is, by definition, $\hat{R}_S$.

(4) Let $\pi_{S,0}$ be the embedding of E_0 into $E_{S,0}$. By Lemma 1, we know that $\text{EVAL}(Q, E_{S,0}) = \pi_{S,0}(\text{EVAL}(Q, E_0))$. Hence, from Lemma 2, we have that $\pi_{S,0}(L) \cap \text{EVAL}(Q, E_{S,0}) = \emptyset$. Now, $\pi_{S,0}(L_S)$ is the set of nodes of $E_{S,0}$ that have a '\$' child. Since none of them belongs to $\text{EVAL}(Q, E_{S,0})$, then the set of content strings of nodes in $\text{EVAL}(Q, E_{S,0})$, which have a '\$' child, has to be empty. □

4. DOMINANCE LOWER BOUND

In this section we discuss the buffering requirements due to evaluation or filtering of queries that have "multi-variate' comparison predicates, i.e., predicates that compare two variables (e.g., `a = b` and `a > b`). We essentially prove that for the equality operators (i.e., `=, !=`), one needs to buffer all the distinct instances of one of the variables, while for the inequality operators (i.e., `<, <=, >, >=`) one needs to store only the instance of maximum or minimum value. The lower bound technique is applicable in a more general setting, to any relational operator R, which has the same existential semantics in XPath as the comparison operators (i.e., R`(a,b)` = `true` if and only if there is an instance x of `a` and an instance y of `b`, so that $R(\text{DATAVAL}(x), \text{DATAVAL}(y)) = \texttt{true}$).

The lower bound we present requires the query to "depend" on the multi-variate predicate; that is, whether the query will select any node from the document or not should depend on the outcome of the predicate's evaluation. This requires us to focus on "redundancy-free" queries [6]. For lack of space, we avoid the treatment of arbitrary redundancy-free queries in this extended abstract, and demonstrate the core ideas of the proof by a lower bound for the evaluation of the specific query $Q =$ `/c[`R`(a,b)]`, where R is any relational operator.

Any binary relation R on a ground set U (i.e., $R \subseteq U \times U$) can be represented as a directed graph G_R. The nodes of G_R are the elements of U; two nodes $u, v \in U$ are connected by an edge (u, v) if and only if $(u, v) \in R$. For example, if R is the "greater than" relation, then the corresponding graph is over the set U of numbers supported by the system (e.g., 64-bit integers), and the edges are as follows: an edge from the maximum number in U to each of the other numbers, an edge from the second largest number to each of the other numbers, except the maximum one, and so forth. If R is the equality relation, then the graph is over all strings/numbers supported by the system, and the only edges are self loops.

For a subset $S \subseteq G_R$ of the graph, we denote by $N(S)$ the *neighborhood* or *dominated set* of S: $N(S) \overset{\text{def}}{=} \{v \in G_R \mid \exists u \in S \text{ s.t. } (u, v) \text{ is an edge in } G_R\}$. Note that two different sets S, S' may have the same neighborhood. We define the *dominance cardinality* of R, denoted $\text{DOM}(R)$, to be the number of distinct neighborhoods, over all possible subsets of G_R. That is, $\text{DOM}(R) = |\{N(S) | S \subseteq G_R\}|$. For example, if R is the "greater than" relation over the set of numbers $U = [1..n]$, there are exactly n distinct neighborhoods (namely, $[1..(n-1)], [1..(n-2)], \ldots, [1..1], \emptyset$), because the neighborhood of any set $S \subseteq U$ equals to the set of numbers that are less than the maximum number in S. If R is the equality relation over a set U, then there are $2^{|U|}$ distinct neighborhoods, because for any set $S \subseteq U$, $N(S) = S$.

A refinement of the dominance cardinality notion is the following: Let R be a relation over U, and let $k \in [1..|U|]$. The *dominance cardinality of R of order k* is the number of distinct neighborhoods of subsets of G_R of size at most k. That is, $\text{DOM}_k(R) = |\{N(S) | S \subseteq G_R, |S| \le k\}|$. For example, if R is the "greater than" relation, then for any k, $\text{DOM}_k(R) = |U|$. On the other hand, if R is the equality relation, then $\text{DOM}_k(R) = \binom{|U|}{k}$.

Given a query Q as above, let $u \in Q$ be the node whose predicate is `R(a,b)`. Let u_a, u_b denote the two children of u corresponding to the node tests `a` and `b`, respectively. For any given document D, we define the *candidate cardinality of D w.r.t. u* to be the maximum number of nodes in D that: (1) are siblings; (2) match u_a and precede any other sibling that matches u_b or vice versa.

Our lower bound is the following:

THEOREM 2. *For the query $Q =$ `/c[`R`(a,b)]`, for every integer k, and for every filtering algorithm $\mathcal{A}$ for Q on XML streams, there exists a document D of candidate cardinality k w.r.t. u on which $\mathcal{A}$ uses $\Omega(\log \text{DOM}_k(R))$ bits of space.*

Note that for the equality operators (`=,!=`), the above lower bound is $\Omega(\log \binom{|U|}{k}) = \Omega(k \log |U|)$ for sufficiently small k's. That means that if R is an equality operator, evaluation of Q on documents that have a node with k children that match u_a (or u_b) would require buffering the distinct data values of these children. On the other hand, when R is an inequality operator (`<,<=,>,>=`), evaluation of Q requires only $\Omega(\log |U|)$ bits of space, which is what is needed to buffer the maximum or minimum data value of the k children that match u_a (or u_b).

PROOF. The proof works by a reduction from the following communication complexity problem $\mathcal{P}$: Alice is given a set S of k elements $a_1, \ldots, a_k$ from U and Bob is given a single element b from U. The goal is to determine whether there is any $i \in [1..k]$ so that $(a_i, b) \in R$. We first prove a lower bound on the one-way communication complexity of this problem:

PROPOSITION 2. $CC_1(\mathcal{P}) = \Omega(\log \text{DOM}_k(R))$.

PROOF. Let Π be any one-way protocol that computes $\mathcal{P}$. Let $A_\Pi(S)$ denote the message Alice sends to Bob, when receiving the input S. Let $B_\Pi(m, b)$ denote the output Bob sends to Alice when receiving a message m from her and an input b.

Let G_R be the graph induced by the relation R. We wish to prove that if $S, S' \subseteq G_R$ are two sets, for which the neighborhoods $N(S), N(S')$ are different, then Alice has to send different messages on S and S'. That would imply that the number of distinct messages Alice can send is $\text{DOM}_k(R)$, and therefore the length of the longest message should be $\Omega(\log \text{DOM}_k(R))$.

Suppose, to reach a contradiction, there exist two sets S, S' of size k, so that $N(S) \neq N(S')$, but $A_\Pi(S) = A_\Pi(S')$. Since $N(S) \neq N(S')$, there exists an element b in $N(S) \setminus N(S')$ or in $N(S') \setminus N(S)$. Assume, for example, that the former holds and that Bob receives b as an input. Since the messages of Alice are the same on S, S', then $B_\Pi(A_\Pi(S), b) = B_\Pi(A_\Pi(S'), b)$, implying that the output of the protocol is the same on both input pairs (S, b) and (S', b). However, $\mathcal{P}(S, b) = \texttt{true}$ while $\mathcal{P}(S', b) = \texttt{false}$, implying that Π makes a mistake. $\square$

The reduction works as follows. Alice, upon receiving the set $S = \{a_1, \ldots, a_k\}$, creates a prefix of a document stream $\overline{\alpha}_S$ as follows: the document has one '`c`' element under the root and k '`a`' elements under the '`c`' element. The data values of the '`a`' elements are $a_1, \ldots, a_k$. The prefix stops after the `endElement` event of the last '`a`' element. Bob, upon receiving his input b, creates a suffix of a document stream $\overline{\beta}_b$ as follows: the document has one '`c`' element under the root and one '`b`' element under the '`c`' element. The '`b`' element has the data value b. The suffix starts at the `startElement` event of the '`b`' element.

It is easy to verify that for any S and b, $\overline{\alpha}_S \circ \overline{\beta}_b$ represents a well-formed document. We denote this document by $D_{S,b}$. It is also immediate that the query will select the 'c' element from $D_{S,b}$ if and only if there exists an $i \in [1..k]$, so that $(a_i, b) \in R$. The candidate cardinality of $D_{S,b}$ w.r.t. u is k, because the k 'a' elements match u_a.

Now, given an algorithm $\mathcal{A}$ that determines whether an XML stream matches Q or not, we can devise the following protocol Π for $\mathcal{P}$: Alice runs $\mathcal{A}$ on the prefix $\overline{\alpha}_S$. When $\mathcal{A}$ is done reading $\overline{\alpha}_S$, Alice sends the state of $\mathcal{A}$ to Bob. Bob resumes the execution of $\mathcal{A}$ on $\overline{\beta}_b$. When the execution is over, he sends 'true' to Alice if $\mathcal{A}$ decided that the document matches the query, and he send 'false' otherwise. From our observation above, Π correctly computes $\mathcal{P}$. The communication cost of Π is the same as the number of bits needed to represent the state of $\mathcal{A}$, which equals to the space of $\mathcal{A}$.

We thus conclude from Proposition 2 that $\mathcal{A}$ has to use $\Omega(\log \text{DOM}_k(R))$ bits of space on at least one of the documents $\{D_{S,b}\}_{S,b}$. $\square$

5. THE ALGORITHM

In this section we describe an XPath streaming algorithm that works for any Forward XPath query over any non-recursive document and almost matches the concurrency lower bound. We start with a general overview of the algorithm and demonstrate the way it works through an example run. We then present a more detailed description of the algorithm, including pseudo-code fragments. We conclude with an overview of the complexity analysis. For lack of space, a full description of the algorithm and its analysis is postponed to the full version of the paper.

Overview and example run In order to demonstrate the core ideas of the algorithm, we will consider the following simple example. Suppose we want to evaluate the query Q = `/a[b > 5]/c` over the following document D:

$\langle a \rangle$ $\langle c \rangle$c1$\langle /c \rangle$ $\langle b \rangle$4$\langle /b \rangle$ $\langle c \rangle$c2$\langle /c \rangle$ $\langle b \rangle$6$\langle /b \rangle$ $\langle b \rangle$3$\langle /b \rangle$ $\langle c \rangle$c3$\langle /c \rangle$ $\langle /a \rangle$.

The official XPath semantics assumes that in the evaluation of a predicate PREDICATE(u) (corresponding to some query node u) over a document node x, every leaf in the expression tree of PREDICATE(u) is evaluated into a *sequence* of data values. Internal nodes are later evaluated over the resulting sequences. In the above example, in the evaluation of the predicate `[b > 5]`, first the sequence $(4, 6, 3)$, corresponding to the data values of the matches to the `b` node, is created. Only then the sequence is compared to the constant 5, and evaluates to `true`, because at least one of its entries is greater than 5.

Note that in the above example the fact that the predicate is going to evaluate to `true` is known already when we encounter the second `b` node in the stream (whose data value is 6). Our algorithm exploits this knowledge and *eagerly evaluates predicates.* Eager evaluation of predicates can save a lot of space. First, we can partially avoid buffering evaluation sequences (in the above example our algorithm will not buffer all the data values of the `b` nodes simultaneously). Second, we can determine earlier which document nodes are selected by the query and thus emit them to the output without buffering them (in the above example we can emit first two `c` nodes to the output as soon as we see the `b` node whose data value is 6. The third `c` node is emitted immediately when encountered).

The principal data structures used by the algorithm are the following: (1) validation array: a boolean array used for checking if the predicate of a given query node has already been satisfied. (2) result buffers: an array of buffers, in which document nodes that may have to be output as part of the result are stored; (3) predicate buffers: an array of buffers, in which document nodes that participate in the evaluation of pending predicates are stored.

The algorithm works by processing the stream of SAX events, taking actions when it receives the `startElement` and `endElement` events for each node.

Figure 4 describes the state of the data structures after each event. At the beginning the validation array for each node is `false` (0) and all buffers are empty. This indicates that none of the predicates have been satisfied yet and that no nodes are being considered as part of the results or for predicate evaluation.

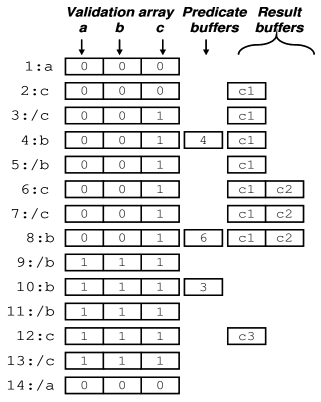


Figure 4: Example run for query `/a[b > 5]/c`

When we see the first `c` (event 2) we have to add it to the result buffers since at this point the predicate `b > 5` is still unverified and thus we do not know whether this `c` will be selected by the query or not. When `c` is closed (event 3) the validation array entry for `c` can be set to `true` (1) since `c` has no predicates to satisfy in the query. When the first `b` arrives (event 4) we start buffering its content in the predicate buffers in order to be able to evaluate the predicate `[b > 5]`. When `b` closes (event 5) we are able to fully evaluate the predicate, which is `false` and therefore the validation array entry for `b` remains unchanged. After the predicate is evaluated, the predicate buffers are discarded. In events 6 and 7 the second `c` is added to the result buffers since the predicate on `b` is still unverified. In event 8 the next `b` occurrence is added to the predicate buffers and in event 9 the predicate on `b` is finally evaluated to `true`. At this point, we turn the validation array entry for `b` to `true`. In addition, since the validation entry for `c` is already `true`, all the constraints on `a` are verified and the `a`'s validation array entry is set to `true` as well. This also allows the `c` nodes that are in the output buffers to be emitted, since we know that they are for sure part of the result set. After they are emitted all the result buffers are discarded. In events 10 and 11 a new `b` node that does not match the predicate is seen. However, even though the predicate evaluation triggered in event 11 returns `false`, we do not reset the validation array entry for `b`. The reason for that is the existential semantics of XPath, that requires the predicate to be valid for just one of the `b` nodes under `a`. When the next `c` arrives in event 12 it is buffered just until `c` closes (event 13). At that point it is emitted as a result and the buffer is discarded. Finally, when the `a` node closes (event 14) the validation array bits are reset. Please note that if events 8 and 9 had not happened, the predicate anchored at `b` would remain `false`, and all the `c` nodes stored in the result buffers would be discarded without being emitted when node `a` closes in event 14.

Detailed description Suppose Q is the input query and D is the input document, given as a stream of SAX events. The algorithm tries to gradually construct matchings of document nodes with the query output node $\textsc{out}(Q)$. Each completed matching results in one document node being emitted to the output.

The algorithm is event-driven. As SAX events arrive, corresponding event handlers are called, updating the global variables of the algorithm. For brevity, we describe below only the handlers for `startElement` and `endElement`.

The algorithm gradually constructs the matchings on a "frontier" of the query. Initially, the frontier consists of the query root alone. When the algorithm receives a `startElement` event of a document node x, it searches for all the nodes u in the frontier, for which x is a "candidate match". For each such node u, the children of u are added to the frontier as well. When the algorithm receives the `endElement` event of x, it removes the children of u from the frontier, and uses them to determine whether x is turned into a "real match" for u or not. The algorithm outputs x if and only if x is found to be a real match for $\textsc{out}(Q)$.

A document node x is a "candidate match" for query node u, if the name of x fits the node test of u and if x relates to the candidate match of $\textsc{parent}(u)$ according to the axis of u (this definition holds for nodes $u \neq \textsc{root}(Q)$). x is also a *real match for* u, if the predicate of u evaluates to `true` on x.

How do we determine whether a document node x is a candidate match for a query node u? In order to do that, we only need to know the name of x and its "document level" (i.e., document depth). By comparing this level to the document level of the candidate match z for $\textsc{parent}(u)$, we know whether x relates to z according to $\textsc{axis}(u)$. Therefore, we can determine whether x is a candidate match for u already at the `startElement` event of u.

On the other hand, determining whether x turns into a real match for u or not requires knowing the string value of x (if u is a leaf) or whether descendants of x are real matches for the children of u. This can be inferred only at the `endElement` event of x.

The algorithm maintains the following global variables. The first five arrays are always of the same size. Each entry in them corresponds to one query node in the frontier.

- `pointerArray`: Pointers to the query nodes in the frontier.
- `IDArray`: Unique IDs of the current candidate matches for the query nodes currently in the frontier.
- `levelArray`: Document levels at to expect candidate matches for the query nodes currently in the frontier, used for processing `child` axis.
- `validationArray`: Boolean flags indicating whether real matches for the query nodes currently in the frontier have already been found.
- `parentArray`: Indices in the above arrays corresponding to the parent of each query node currently in the frontier.
- `predicateArray`: Contents of document nodes that are needed for evaluating predicates of query nodes in the frontier.
- `resultArray`: Contents of document nodes that are candidate matches for $\textsc{out}(Q)$ and it is not yet clear whether they will turn into real matches.

In addition, the variable `nextIndex` contains the size of the first five arrays, `nextPred` contains the size of `predicateArray` and `nextResult` contains the size of `resultArray`.

At initialization, the query root is inserted to `pointerArray`, its `levelArray` entry is set to 0, its `validationArray` entry is set to `false`, and its `parentArray` entry is set to NULL. The variables `nextIndex`, `nextPred`, and `nextResult` are set to 0 and the arrays `predicateArray` and `resultArray` are left empty.

The `startElement` event handler is called every time a new document node x starts. The function iterates over all the query nodes u in the frontier, for which x is a candidate match (lines 4-7). In lines 8-9, we distinguish between treatment of query nodes along the succession path of the query root (the "main path") and ones that are not. The reason is the following. For nodes along the main path, we wish to find all possible matches in the document, because these may turn into distinct results in the output. On the other hand, nodes that do not belong to the main path are necessarily part of predicates. For predicate evaluation, we do not really need to find *all* possible matches: it suffices to find at least *one* good match (due to the existential semantics of XPath). For example, if $Q =$ `/a[b > 5]/c`, then we will look for all the matches to the `c` node, but we will stop looking for matches for the `b` node as soon as we find a match whose data value is greater than 5.

If u is an internal node, checking whether x turns into a real match or not will require finding real matches for the children of u in the subtree rooted at x. We thus insert all the children of u into the frontier (lines 11-19). Each match is assigned a unique ID (lines 16-17). In lines 20-22 we distinguish between the setting of the validation array bits for nodes that are part of predicate expressions and nodes that are part of the main path, i.e. outside of predicate expressions. The validation array entry for nodes used in predicate evaluation is used for predicate evaluation and is therefore existential. Contrary to this, the validation array bits of the main path nodes are used to determine if the result buffer can be emitted. For this purpose, we need to know if the currently opened document nodes matching the nodes on the main path have satisfied all conditions. Therefore the validation of the nodes on the main path are reset each time we open an element matching this node. Finally in lines 23-33 we add the current event to the predicate array if this node is the last node in a path used in a predicate expression; similarly for nodes that match the last step of the main path we add an entry to the result array.

Function `endElement` is called once for every close element event in the document stream. It starts by decrementing the current level (line 1). It then checks if there are nodes in the global arrays that need to be removed since their parent is

```
 1:Function startElement(x)
 2:  maxIndex := nextIndex - 1
 3:  for i := 0 to maxIndex do
 4:    u := pointerArray[i]
 5:    if ((ntest(u) = label(x)) or (ntest(u) = *)) then
 6:      if ((axis(u) = descendant) or
 7:          (levelArray[i] = currentLevel)) then
 8:          if (validationArray[i] = TRUE and !isMainPath(u))
 9:            continue
10:          end if
11:          for c in children(u) do
12:            parentArray[nextIndex] := i
13:            pointerArray[nextIndex] := c
14:            levelArray[nextIndex] := currentLevel + 1
15:            validationArray[nextIndex] = FALSE
16:            IDArray[nextIndex] := nextID
17:            nextID := nextID + 1
18:            nextIndex := nextIndex + 1
19:          end for
20:          if (isMainPath(u))
21:            validationArray[i] = FALSE
22:          end if
23:          if (isPredicateTerm(u) and validationArray of
24:              the node that anchors u's predicate is FALSE)
25:            predicateArray[nextPred].value := x
26:            predicateArray[nextPred].id := IDArray[i]
27:            nextPred := nextPred + 1
28:          end if
29:          if (isResult(u))
30:            resultArray[nextResult].value := x
31:            resultArray[nextResult].id := IDArray[i]
32:            nextResult := nextResult + 1
33:          end if
34:      end if
35:    end if
36:  end for
37:  currentLevel := currentLevel + 1
38:end startElement
```

```
 1:Function endElement()
 2:  currentLevel := currentLevel - 1
 3:  i := nextIndex - 1
 4:  while (levelArray[i] > currentLevel) do
 5:    nextIndex := nextIndex - 1
 6:    i := i - 1
 7:  end while
 8:  for i := 0 to nextIndex do
 9:    u := pointerArray[i]
10:    if ((ntest(u) = label(x)) or (ntest(u) = *)) then
11:      if ((axis(u) = descendant) or
12:          (levelArray[i] = currentLevel)) then
13:          if (hasPredicate(u) and validationArray[i] = FALSE)
14:            validationArray[i] := evalPred(u)
15:            remove buffers from the predicateArray
16:          else if (isLeaf(u)) then
17:            validationArray[i] := TRUE
18:          else if (validationArray[i] = FALSE) then
19:            c := validationArray bits for children(u)
20:            validationArray[i] := c[0]
21:          end if
22:          if (isPredicateTerm(u) and validationArray[i]) then
23:            purge buffers based on operator properties and
24:            try to eagerly evaluate the predicate
25:          end if
26:          if (validationArray[i] = FALSE) then
27:            remove buffers from the resultArray
28:          else if (all validationArray bits for the nodes in
29:                   in the main path are TRUE) then
30:            output the results
31:            remove buffers from the resultArray
32:          end if
33:      end if
34:    end if
35:  end for
36:end endElement
```

the node being closed (lines 2-7). `endElement` then updates the validation array entries for the nodes being closed (lines 13-21). If the node being closed has a predicate (lines 13-15) the predicate is evaluated by invoking `evalPred`. Function `evalPred` simply evaluates the predicate tree anchored at the matched query node and returns `true` if the predicate is valid and `false` otherwise. In order to do the predicate evaluation `evalPred` may need to access the predicate buffers. After the predicate evaluation is done the predicate buffers are discard (line 15). If the node being closed is a leaf, the validation array is set to `true` since it does not have any constraints that still need to be verified (lines 16-17). Finally, if the node being closed is an internal node that has no predicate (lines 18-20), it must have only one child node. Therefore its validation array entry is set to `true` only if the all the constraints in the child node have been satisfied, i.e., the validation array entry for the child node is `true`. Please note that in order to enforce the existential semantics of XPath we just update the validation array entry for the closing node if it is not already set to `true`. If the node being closed is part of a predicate we try to eagerly evaluate that predicate (lines 22-25). For example, in query `a[b > 5]/c`, when we close b we try to eagerly evaluate the predicate anchored at a. This eager evaluation allows for verifying predicates as soon as possible, which in turn allows us to emit results and discard buffers as soon as possible. Just before the eager evaluation we purge the buffer array from entries that are not needed, based on the operator properties. For example, for non-equality comparison we preserve only the maximum/minimum value. After all the predicates have been evaluated and all validation array entries have been set we check if we can emit results and discard the result buffers (lines 26-31). If the validation array entry for the closing node is `false` we can discard all the result buffers seen after the closing node (lines 26-27). Otherwise we test if all the query constraints have been satisfied, in which case we output all the results buffered so far and discard their buffers (lines 28-31).

Analysis The correctness of the algorithm follows from the fact it outputs all the matches for the query output node. A rigorous analysis is postponed to the full version of the paper. Since the document on which the algorithm runs is non-recursive, we are guaranteed that at any given step of the algorithm, the frontier consists of at most one copy of each query node. Thus the size of the arrays that hold the frontier cannot exceed $|Q|$. Each entry in these arrays requires at most logarithmic space. Furthermore, document nodes are kept in the result array only as long as we have no information to decide whether they should belong to the output or not. Therefore, the number of entries in the result array is at most the concurrency $\textsc{concur}(D, Q)$. The size of each entry may be large though, because we need to record the content of each candidate result. Various compression techniques may be used to minimize the size of these entries. We conclude that the space complexity of the algorithm is as follows:

THEOREM 3 (SPACE COMPLEXITY). *For every Forward XPath query Q and for every non-recursive document D, the algorithm, when running on Q and D:*

1. *Uses at most $O(|Q|(\log(|Q|)+\log(|D|)))$ space for storing the frontier.*
2. *Uses at most $O(\textsc{concur}(D, Q))$ for the result array.*

As for the running time, the handlers of the `startElement` and `endElement` events iterate over all nodes in the frontier, and spend a fixed amount of time in each. Therefore:

THEOREM 4 (TIME COMPLEXITY). *For every Forward XPath query Q and for every non-recursive document D, the algorithm, when running on Q and D, terminates in $\tilde{O}(|Q| \cdot |D|)$ steps (where the $\tilde{O}$ notation suppresses polylogarithmic factors).*

6. CONCLUSIONS

Buffering may constitute a major memory bottleneck for XPath processing over XML streams. In this paper we identified two sources of this high memory consumption: (1) full fledged evaluation of queries with predicates and (2) evaluation of multi-variate predicates. We presented lower bounds for these two scenarios and described an algorithm that gets close to these lower bounds for a class of XPath queries. A future research direction is to extend our memory requirements study to evaluation of XQuery queries. XQuery has additional features, such as multiple output nodes for a single query, which possibly necessitate high memory usage.

7. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. 26th VLDB*, pages 53–64, 2000.

[3] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. 21st PODS*, pages 221–232, 2002.

[4] I. Avila-Campillo, T. J. Green, A. Gupta, M. Onizuka, D. Raven, and D. Suciu. XMLTK: An XML toolkit for scalable XML stream processing. In *Proc. 1st Workshop on Programming Languages for XML (PLAN-X)*, 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st PODS*, pages 1–16, 2002.

[6] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. In *Proc. 23rd PODS*, pages 177–188, 2004.

[7] Z. Bar-Yossef, T. S. Jayram, R. Kumar, and D. Sivakumar. An information statistics approach to data stream and communication complexity. In *Proc. 43rd FOCS*, pages 209–218, 2002.

[8] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Siméon. *XML Path Language (XPath) 2.0.* W3C, `http://www.w3.org/TR/xpath20`, 2004.

[9] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proc. 18th ICDE*, pages 235–244, 2002.

[10] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proc. 18th ICDE*, pages 341–342, 2002.

[11] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[12] M. Fernández, A. Malhotra, J. Marsh, M. Nagy, and N. Walsh. *XQuery 1.0 and XPath 2.0 Data Model.* W3C, `http://www.w3.org/TR/xpath-datamodel`, 2004.

[13] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. 22nd PODS*, pages 179–190, 2003.

[14] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proc. 9th ICDT*, pages 173–189, 2003.

[15] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proc. 22nd SIGMOD*, pages 419–430, 2003.

[16] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, volume 50, pages 107–118, 1999.

[17] P. Indyk and D. Woodruff. Tight lower bounds for the distinct elements problem. In *Proc. 44th FOCS*, pages 283–289, 2003.

[18] Z. Ives, A. Levy, and D. Weld. Efficient evaluation of regular path expressions on streaming XML data. Technical report, University of Washington, 2000.

[19] V. Josifovski, M. Fontoura, and A. Barta. Querying XML streams. *The VLDB Journal*, 2004. to appear.

[20] C. Koch, S. Scherzinger, N. Schweikardt, and B. Stegmaier. Schema-based scheduling of event processors and buffer minimization for queries on structured data streams. In *Proc. 30th VLDB*, pages 228–239, 2004.

[21] E. Kushilevitz and N. Nisan. *Communication Complexity.* Cambridge University Press, 1997.

[22] A. Malhotra, J. Melton, and N. Walsh. *XQuery 1.0 and XPath 2.0 Functions and Operators.* W3C, `http://www.w3.org/TR/xquery-operators`, 2004.

[23] A. Marian and J. Siméon. Projecting XML documents. In *Proc. 29th VLDB*, pages 213–224, 2003.

[24] S. Muthukrishnan. Data streams: Algorithms and applications. In *Proc. 14th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, page 413, 2003. `http://www.cs.rutgers.edu/~muthu/stream-1-1.ps`.

[25] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proc. 19th ICDE*, pages 702–704, 2003.

[26] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proc. 22nd SIGMOD*, pages 431–442, 2003.

[27] M. Saks and X. Sun. Space lower bounds for distance approximation in the data stream model. In *Proc. 34th STOC*, pages 360–369, 2002.

[28] L. Segoufin. Typing and querying XML documents: Some complexity bounds. In *Proc. 22nd PODS*, pages 167–178, 2003.

[29] D. Woodruff. Optimal space lower bounds for all frequency moments. In *Proc. 15th SODA*, pages 167–175, 2004.

[30] A. C.-C. Yao. Some complexity questions related to distributive computing. In *Proc. 11th STOC*, pages 209–213, 1979.