On the Memory Requirements of XPath Evaluation over XML Streams

Ziv Bar-Yossef
IBM Almaden
650 Harry Road
San Jose, CA 95120, USA.
ziv@almaden.ibm.com

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

The important challenge of evaluating XPath queries over XML streams has sparked much interest in the past two years. A number of algorithms have been proposed, supporting wider fragments of the query language, and exhibiting better performance and memory utilization. Nevertheless, all the algorithms known to date use a prohibitively large amount of memory for certain types of queries. A natural question then is whether this memory bottleneck is inherent or just an artifact of the proposed algorithms.

In this paper we initiate the first systematic and theoretical study of lower bounds on the amount of memory required to evaluate XPath queries over XML streams. We present a general lower bound technique, which given a query, specifies the minimum amount of memory that *any* algorithm evaluating the query on a stream would need to incur. The lower bounds are stated in terms of new graph-theoretic properties of queries. The proof is based on tools from communication complexity.

We then exploit insights learned from the lower bounds to obtain a new algorithm for XPath evaluation on streams. The algorithm uses space close to the optimum. Our algorithm deviates from the standard paradigm of using automata or transducers, thereby avoiding the need to store large transition tables.

1. INTRODUCTION

XML [5] is quickly gaining dominance as a format for exchanging and storing semi-structured data. The most popular language for querying XML data is XPath [10], which is part of both XSLT [9] and XQuery [4], the two WWW Consortium language standards for querying and transforming XML. XPath allows addressing portions of XML documents based on their structure and data values.

Recently, several algorithms for evaluating XPath queries over XML streams have been proposed [1, 3, 6, 11, 14, 15, 16, 17, 19, 20]. These algorithms evaluate the query using a one-pass sequential scan of the XML document, while keeping only small critical portions of the data in main memory for later use. Streaming algorithms are the prime choice for domains where the XML documents are transferred between systems. Due to their predictable access pattern, they are also efficient over pre-stored XML data.

While demonstrating a steady progress, both in terms of the scope of the fragment of XPath supported and in terms of the time and space complexity, even the state of the art algorithms incur high memory costs on certain types of queries. Anecdotal evidence of this phenomenon has been recorded before [14, 20], however to date there has not been any formal or systematic study of the source for the high memory costs.

This paper lays the first theoretical foundations for *lower bounds* on the amount of memory required to evaluate XPath queries over XML streams. We introduce powerful lower bound techniques, based on the theory of communication complexity [23]. As opposed to previous results in the area [14], our lower bounds hold for *any* algorithm, not for a specific algorithm or for a restricted class of algorithms. Our lower bounds are also *not* anecdotal - we are not providing examples of queries that incur large memory costs. Instead, we introduce a technique that can assert the memory needed to evaluate *any given query* within a subset of the XPath language described below.

The lower bounds hold even for the weaker task of *filtering* a sequence of streaming XML documents based on whether they match a given XPath query. In order to show that the bounds are tight, we designed a new filtering algorithm, which is particularly memory-efficient while not suffering a significant loss in running time. To the best of our knowledge, this algorithm has the best theoretical efficiency guarantees among all the known algorithms for filtering streaming XML documents. We note that the algorithm could be extended to provide also a full-fledged evaluation of XPath queries [17].

Complexity measures Our lower bounds apply to a very strong measure of complexity, which we call the *instance data complexity* and explain in more detail next. Any query language is associated with an *evaluation function* Φ which maps query-database pairs (Q, D) into output values. By fixing a query Q, we get an induced mapping Φ_Q from databases to output values. Similarly, by fixing a database D, we obtain an induced mapping Φ_D from queries to output values. Vardi [22] defined three standard measures of complexity for database query languages: the *data complex-*

PODS 2004, June 14–16, 2004, Paris, France.

ity (the complexity of Φ_Q, for the worst-case choice of Q), the *expression complexity* (the complexity of Φ_D, for the worst-case choice of D), and the *combined complexity* (the complexity of Φ). These measures are typically given in terms of the input size.

In this paper we prove lower bounds on stronger measures of complexity, which are reminiscent of the notion of *instance-optimality* [12]. Rather than considering the standard data complexity, which is a worst-case measure, we study the *instance data complexity*. Formally, we characterize the complexity of *each one* of the mappings $\{\Phi_Q\}_Q$. Naturally, for different queries, the corresponding mappings may have different complexities. Thus, the complexity of Φ_Q is given in terms of quantitative properties of Q as well as in terms of parameters of the input database. For the latter, we consider other parameters than the database size, such as the document depth and the document recursion depth.

Since characterizing the complexity of all the mappings Φ_Q is typically very hard, we usually have to restrict the class of queries Q for which we can get such a characterization. Thus, each of our lower bound theorems is accompanied by a definition of a fragment $\mathcal{F}$ of the query language. The theorem then bounds the complexity of Φ_Q for all $Q \in \mathcal{F}$.

Our results We prove three memory lower bounds. The main lower bound is stated in terms of a new graph-theoretic property of queries, which we call the *query frontier size*. When viewed as a rooted tree, the frontier of a query Q at a node $u \in Q$ is the collection of u's siblings and of its ancestors' siblings. The frontier size of Q is the size of the largest frontier, over all nodes $u \in Q$. We prove that for any Q belonging to a large fragment of XPath, the query frontier size of Q is a lower bound on the space complexity of evaluating Φ_Q on XML streams.

The query frontier size is always at most linear in the size of the query. For "balanced" queries, it is even logarithmic in the size of the query (proportional to the product of the fan out and the depth of the tree). This lower bound is thus very far from the worst-case exponential upper bounds of many of the current algorithms for XPath streaming evaluation [14, 15, 19, 20]. All of these algorithms are based on finite state automata, and the exponential blowup in memory is largely due to the loss incurred by simulating nondeterministic automata by deterministic ones. Our upper bounds (discussed below) show that in fact this exponential loss is not necessary and the truth lies much closer to the lower bounds we present in this paper.

There are some restrictions on the queries to which this lower bound is applicable. The most important of these is that the query should be "subsumption-free" (see definition in Section 4), which intuitively means that the query has no redundancies. Usually, by rewriting the query into a minimal form, one can get an equivalent subsumption-free query, to which the lower bound applies. We note, however, that such a rewrite may be NP-hard in the general case.

Our second lower bound is in terms of the document *recursion depth*. A document is called recursive, if it contains nodes that are nested within each other, and that match the same query node. The recursion depth of the document is the length of the longest sequence of such nodes. We show that evaluating XPath queries on streaming XML documents of recursion depth r requires $\Omega(r)$ space.

Our last lower bound is in terms of the *document depth*. We show that $\Omega(\log d)$ space is needed to evaluate XPath queries on streaming XML documents of depth d. Note that this lower bound is incomparable with the recursion depth lower bound, because the recursion depth r can be anywhere between 1 and d.

In the second part of the paper we present an XML filtering algorithm (cf. [1]) that supports a large fragment of the XPath language, including predicates, descendant axes, and wildcard node tests. Given an XML document D and an XPath query Q, it determines whether D matches Q (i.e., the evaluation of Q on D is non-empty). We show that the memory used by the algorithm matches the lower bounds modulo logarithmic factors for many XPath queries. The proposed algorithm builds on our recent XQuery evaluation algorithm for XML streams [17], which avoids the finite state automata paradigm used by the rest of the known algorithms, and thereby is able to achieve significant savings in space. The novelty in the current paper is a more sophisticated manipulation of the global data structures, which reduces the memory consumption to close to the query frontier size lower bound.

We note that our algorithm satisfies the very strong notion of "instance-optimality" [12]—for *every* query instance Q (belonging to an appropriate fragment of XPath), the algorithm is doing better than any other algorithm. This is much stronger than the standard worst-case optimality.

The rest of the paper is organized as follows. Section 2 overviews related work. In Section 3 we provide background material on XML and XPath, streaming algorithms, and communication complexity. Section 4 defines the various XPath fragments used in the paper. In Section 5 we describe and prove the three lower bounds. In Section 6 we outline the new filtering algorithm. We conclude in Section 7 with directions for future research.

2. RELATED WORK

As noted earlier, several streaming algorithms have been proposed for varying fragments of XPath and XQuery [1, 3, 6, 11, 14, 15, 16, 17, 19, 20]. While some complexity analysis is provided with most of these algorithms, none of them presents a systematic study of lower bounds as we do. Most of these algorithms are based on finite-state automata, whose number of states is exponential in the query size in the worst-case. Our algorithm, on the other hand, uses $\tilde{O}(|Q| \cdot r)$ space and $\tilde{O}(|Q| \cdot |D| \cdot r)$ time, where $|Q|$ is the query size, $|D|$ is the document size, r is the document recursion depth, and $\tilde{O}$ suppresses logarithmic factors.

Gottlob, Koch, and Pichler [13] and Segoufin [21] studied the complexity of evaluating XPath queries over (not necessarily streaming) XML documents. They showed that a large fragment of the XPath language, called Core-XPath, is P-complete w.r.t. combined complexity, while smaller fragments are LOGCFL-complete and NL-complete. They also showed that XPath is L-hard under AC^0-reductions w.r.t. data complexity. The differences from our work are: (1) we consider evaluation of XPath over XML streams, and thus are able to derive stronger lower bounds for this special case; and (2) we prove lower bounds on the instance data complexity and not on the worst-case data or combined complexities.

Choi, Mahoui, and Wood [7, 8] consider memory lower

```
Path := /Step | //Step | Path Path
Step := NodeTest | NodeTest '[' Expression ']'
NodeTest := ElementTest | AttributeTest |
            text(Constant) | *
PathOrConstant := Path | Constant
Expression := Path |
              PathOrConstant Operator  PathOrConstant|
              Expression and Expression |
              Expression or Expression |
              not(Expression)
```

Operators can be any valid XPath operator. Constants are string and numeric constants.

Figure 1: Grammar of XPath fragment considered in this paper.

bounds for evaluating XPath queries over streams of *indexed* XML data. Thus, in their setting the input is not a single stream consisting of an XML document, but rather a collection of streams (generated in a pre-processing step from the XML data), each of which consists of all the XML elements that share a certain label. We prove lower bounds on the direct evaluation of XPath queries on (non-processed) streaming XML documents.

Arasu *et al.* [2] prove space lower bounds for the evaluation of continuous select-project-join queries over relational data streams. While our setting is completely different, some of the challenges encountered are similar. In particular, both papers consider instance data complexity. We note, however, that their goals were much more coarse-grained: separating between queries Q for which Φ_Q has constant ("bounded") space complexity and ones that have unbounded space complexity. We give a finer estimation of the space complexity of Φ_Q, for all Q.

3. PRELIMINARIES

XML and XPath An XML [5] document is a rooted labeled tree, in which the children of each node are ordered. The label of a node x, denoted $\text{LABEL}(x)$, is a symbol from an alphabet Σ. The label of the root is always the special symbol \$. Each node of the tree is additionally associated with a value α taken from the domain of all finite-length strings $\mathcal{V}$. The string value of any node x, denoted $\text{TEXT}(x)$, is defined recursively to be the concatenation of its own value with the values of its children.

Figure 1 describes *Forward XPath*: the fragment of XPath that consists of queries that have only child, attribute, or descendant axes. All the XPath fragments considered in this paper are subsets of Forward XPath (see Section 4).

We use the following combinatorial view of XPath queries [17]: a query Q is a rooted labeled tree, whose nodes correspond to the location steps of the query. Each node $u \in Q$ has the following "data members": (1) a *node test* label, denoted $\text{NTEST}(u)$, which is either a symbol from Σ or the wildcard $*$ (the node test of the root is always \$); (2) a predicate P_u; (3) at most one *successor child* and 0 or more *predicate children*; (4) an *axis*, which is either "child", "attribute", or "descendant" (for the rest of the paper, we ignore the attribute axis, since it can be handled in the same way as the child axis). Figure 2 shows an example query tree. The successor child is the following "step" in the XPath expression, and is marked in Figure 2 by a dashed line. For example, the successor child for a is the left-hand-side b, since it is the node in the next step (after the predicate for a). Each node is annotated with an XPath axis. In Figure 2 we use '/' to indicate child axis and '//' for descendant axis.

The predicate is an expression tree, whose output is always Boolean. Internal nodes of the tree are labeled either by Boolean connectors (AND,OR,NOT) or by operators (such as $<, >, \leq, \geq, =$). Leaves are labeled either by constant values from the domain $\mathcal{V}$ or by pointers to the predicate children of u.

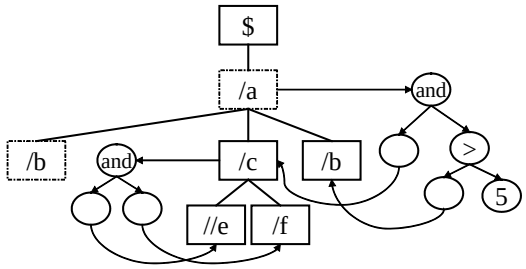

Figure 2: Query tree for `/a[c[.//e and f] and b > 5]/b`

Throughout, we will use the letters x, y, z to denote document nodes and the letters u, v, w to denote query nodes.

Query evaluation We next follow the XPath specification [10] in defining how one evaluates an XPath query Q on an XML document D.

For a query node $u \in Q$, we define the regular path expression corresponding to u, ϕ_u, by induction. If u is the root, then $\phi_u = \$$. Otherwise, let v be the parent of u and let a be its node test. If u has a child axis and if $a \neq *$, then $\phi_u = \phi_v a$. If $a = *$, then $\phi_u = \phi_v \diamond$, where $\diamond$ is a wildcard that can be matched by any symbol in Σ. If u has a descendant axis and $a \neq *$, then $\phi_u = \phi_v \diamond^* a$. If $a = *$, then $\phi_u = \phi_v \diamond^+$.

We will denote by $\text{PATHS}(u)$ the language accepted by this regular expression. For a document node x, we denote by $\text{LPATH}(x)$ the sequence of labels on the path from the root to x. We say that x *structurally matches* u, if $\text{LPATH}(x) \in \text{PATHS}(u)$.

Let x be a document node that structurally matches a query node u. For every child v of u, $\text{SMATCH}_{u=x}(v)$ denotes the set of document nodes y that: (1) structurally match v; and (2) are children of x, if v has a child axis, or are descendants of x, if v has a descendant axis.

The *evaluation of u on x*, denoted $\text{EVAL}(u, x)$, is a subset of $\mathcal{V}$, and is defined recursively. Let r be the root of the predicate tree P_u. If x does not structurally match u or if $\text{PEVAL}(r, x) = \emptyset$ (see a definition of $\text{PEVAL}(\cdot,\cdot)$ below), then $\text{EVAL}(u, x) = \emptyset$. Otherwise, if u has a successor v, then $\text{EVAL}(u, x) = \bigcup_{y \in \text{SMATCH}_{u=x}(v)} \text{EVAL}(v, y)$. Else, $\text{EVAL}(u, x) = \{\text{TEXT}(x)\}$. We say that x *fully matches* (or simply *matches*) u, if $\text{EVAL}(u, x) \neq \emptyset$.

The evaluation of a node $s \in P_u$ on x, denoted $\text{PEVAL}(s, x)$, is also a subset of $\mathcal{V}$ and is defined as follows. (1) If s is labeled by a constant t, then $\text{PEVAL}(s, x) = \{t\}$. (2) If s

is labeled by a pointer to a predicate child v of u, then $\text{PEVAL}(s,x) = \bigcup_{y \in \text{SMATCH}_{u=x}(v)} \text{EVAL}(v,y)$. (3) If s is labeled by an operator R of arity k, and if the children of s are $r_1, \ldots, r_k$, then $\text{PEVAL}(s,x) = \{R(t_1, \ldots, t_k) \mid t_i \in \text{PEVAL}(r_i, x), i = 1, \ldots, k\}$. (4) If s is labeled by a Boolean connector (i.e., AND, OR, NOT) C of arity k, and if $r_1, \ldots, r_k$ are the children of s, then

$$\text{PEVAL}(s,x) = C(\text{PEVAL}(r_1, x), \ldots, \text{PEVAL}(r_k, x)).$$

The arguments to C are treated as `true` if and only if they represent non-empty sets.

The evaluation of a query Q on a document D, denoted $\Phi(Q,D)$, equals to $\text{EVAL}(r_Q, r_D)$, where r_Q is the root of Q and r_D is the root of D. We say that D *matches* Q, if $\Phi(Q,D) \neq \emptyset$.

XML streams An XML document can be viewed as an event stream of the following types: (1) `startDocument()` (also denoted $\langle\$\rangle$), (2) `endDocument()` (also denoted $\langle/\$\rangle$), (3) `startElement(a)` (also denoted $\langle a\rangle$), (4) `endElement(a)` (also denoted $\langle/a\rangle$), and (5) `text(`α`)`. a belongs to Σ and α belongs to $\mathcal{V}$.

Although the streaming algorithm can only access the document sequentially, there are no restrictions on the access to the query.

Communication complexity In the communication complexity model [23, 18] two players, Alice and Bob, jointly compute a function $f : \mathcal{A} \times \mathcal{B} \to \mathcal{Z} \cup \{\perp\}$. Alice is given $\alpha \in \mathcal{A}$ and Bob is given $\beta \in \mathcal{B}$, and they exchange messages according to a protocol. If $f(\alpha,\beta) \neq \perp$, 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.

Lemma 1 below shows that for any function $g : \mathcal{X}^* \to \mathcal{Z} \cup \{\perp\}$, $CC(g')$ is a lower bound on the space complexity of g in the streaming model, where g' is a two-argument function obtained from g.

For any integer $k \geq 2$, we define g^k to be a two-argument function induced by g. Inputs of g^k are obtained by all the possible partitions of inputs of g into k consecutive segments (of possibly varying lengths). Given an input x of g and a partition of x into k segments, we denote by $\alpha_1, \ldots, \alpha_p$ the odd segments and by $\beta_1, \ldots, \beta_q$ the even segments ($p = \lceil k/2 \rceil$ and $q = \lfloor k/2 \rfloor$). $\alpha = (\alpha_1, \ldots, \alpha_p)$ is the first input argument of g^k and $\beta = (\beta_1, \ldots, \beta_q)$ is its second input argument.

LEMMA 1 (REDUCTION LEMMA). *For any function $g : \mathcal{X}^* \to \mathcal{Z} \cup \{\perp\}$ and for any integer $k \geq 2$, any streaming algorithm computing g requires at least $CC(g^k)/(k-1)$ bits of memory.*

The proof is rather standard (cf. [18]), but is provided below for completeness.

PROOF. Let M be any streaming algorithm computing g, and let S be the space used by M. We will show how to use M to construct a protocol that computes g^k with $(k-1) \cdot S$ bits of communication. It would then immediately follow that $S \geq CC(g^k)/(k-1)$.

Recall that g has two input arguments: α and β, where $\alpha = (\alpha_1, \ldots, \alpha_p)$ and $\beta = (\beta_1, \ldots, \beta_q)$, and by interleaving the entries of these two vectors one obtains an input stream $x \in \mathcal{X}^*$ for the function g, so that $g(x) = g^k(\alpha,\beta)$.

The protocol for g^k works as follows. Alice starts by running the streaming algorithm M on α_1. When she gets to the end of α_1 she sends to Bob the current state of the algorithm M. Note that the description of this state requires at most S bits. Bob can now continue the execution of M on β_1. When he gets to the end of β_1, he sends the reached state of M back to Alice, who continues the execution on α_2. Alice and Bob keep on in this manner, until one of them (say, Alice) gets to the end of the execution of M. Alice then sends whatever M outputs.

It is rather obvious that this protocol indeed computes g^k correctly. The number of messages exchanged between Alice and Bob is exactly $p + q - 1 = k - 1$, and the length of each message is at most S bits. Thus the total communication of the protocol is $S \cdot (k-1)$. □

Lemma 1 thus reduces the task of proving space lower bounds for streaming algorithms to the task of proving communication complexity lower bounds. For the latter a rich set of techniques is available. We will mainly capitalize on the fooling set technique, which we describe next.

Definition 1. Let $f : \mathcal{A} \times \mathcal{B} \to \mathcal{Z} \cup \{\perp\}$ be a function. A *fooling set* for f is a subset S of the inputs, which satisfies: (1) all the inputs in S are well-formed and share the same output value z; and (2) for any two distinct inputs (α,β) and (α',β') in S, either (α,β') is well-formed and $f(\alpha,\beta') \neq z$ or (α',β) is well-formed and $f(\alpha',\beta) \neq z$.

THEOREM 1 (FOOLING SET TECHNIQUE). *Let S be any fooling set for f. Then, $CC(f) \geq \log |S|$.*

The proof appears in Chapter 1 of [18], but we provide it here for completeness:

PROOF. Let Π be any protocol that computes f. Let $\pi_{\alpha,\beta}$ be the transcript of messages exchanged between Alice and Bob when they execute Π and are given the inputs α and β, respectively. We will show that for any two distinct inputs (α,β) and (α',β') in S, $\pi_{\alpha,\beta}$ and $\pi_{\alpha',\beta'}$ must be different. It would then follow that Π has at least $|S|$ different transcripts, and thus the length of at least one of them has to be at least $\log |S|$.

Assume, to the contradiction, that there are inputs (α,β) and (α',β') in S so that $\pi_{\alpha,\beta} = \pi_{\alpha',\beta'} = \pi$. Since both inputs are well formed and share the same output value z, the last message in π must be z.

Consider now the inputs (α,β') and (α',β). It is not hard to prove, by induction on the number of messages in π, that π must be also the transcript on these inputs. It follows that Π outputs the value z on both inputs. However, we know that at least one of them is a well-formed input whose output value should be different from z. Thus, Π makes an error on this input, which is a contradiction to its correctness. □

4. XPATH FRAGMENTS

In the following we define the principal fragments of Forward XPath considered in this paper. Figure 3 illustrates the connections among these fragments.

Symmetric XPath A predicate P_u is called *symmetric* if its evaluation on any document node x is independent of the order of x's children. For example, the predicates

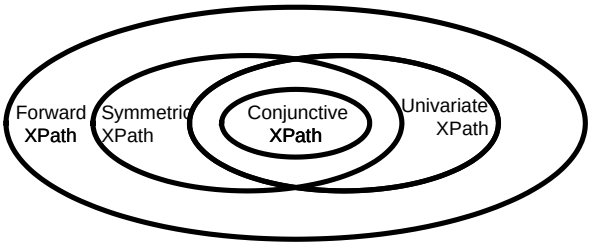

Figure 3: XPath fragments used in the paper

in `a[b]`, `a[b > 5]`, and `a[b > c]` are symmetric, while the predicates in `a/b[1]` and `a/b[last()]` are not. Typically, if a predicate consists of "positional" function operators (such as `position()` and `last()`), then it is not symmetric. *Symmetric XPath* is the fragment of Forward XPath consisting of the queries all of whose predicates are symmetric.

Univariate XPath An *atomic predicate* is a maximal subexpression of a predicate that does not consist of Boolean connectors. Any predicate can be described as a Boolean formula over atomic predicates. An atomic predicate is called *univariate*, if it depends on the value of at most one node in the query tree. For example, `a > 5` and `b` are univariate atomic predicates, while `a > b` is not. *Univariate XPath* is the fragment of Forward XPath consisting of the queries all of whose atomic predicates are univariate.

Subsumption-free XPath The subset of queries of Univariate XPath that are in addition *subsumption-free* (see definition below) is called *Subsumption-free XPath* (not shown in Figure 3).

Let u be any node in a query that belongs to Univariate XPath. If u is used by a univariate atomic predicate, we denote by $\text{SAT}(u)$ the set of *satisfying values* for u—these are the values that make the atomic predicate `true`. Otherwise, $\text{SAT}(u) = \mathcal{V}$. For example, the set of satisfying values for a node labeled `a` in the atomic predicate `a > 5` is $(5, \infty)$.

The set of *full matches* for u, denoted $\text{MATCHES}(u)$, is the product set $\text{PATHS}(u) \times \text{SAT}(u)$. u is said to *structurally subsume* another node v, if $\text{PATHS}(u) \subseteq \text{PATHS}(v)$. u is said to *fully subsume* v, if $\text{MATCHES}(u) \subseteq \text{MATCHES}(v)$. In the example query (Figure 2) the two nodes labeled b structurally subsume each other; however, the right-hand-side b node fully subsumes the left-hand-side one, but not vice versa.

Definition 2. Let L be the set of leaf nodes in a query Q. Q is called *subsumption-free*, if for all nodes $u \in Q$, $\text{MATCHES}(u)$ is not a subset of $\bigcup_{v \in L, v \neq u} \text{MATCHES}(v)$.

Subsumption-free queries are intuitively queries that do not contain "redundancies". Some queries can be rewritten to be subsumption-free, by eliminating redundant portions. In general, however, we do not know whether every query has an equivalent subsumption-free query. Furthermore, even given a query that has a subsumption-free form, it may be NP-hard to find this form.

The query `/a[(b > 3) and (b > 5)]` is not subsumption-free, because the right-hand-side `b` subsumes the left-hand-side one. It can be easily made subsumption-free by removing the left-hand-side `b`, resulting in the query `/a[b > 5]`. An example of a query which is not subsumption-free but also cannot be made subsumption-free simply by removing one of its nodes is depicted in Figure 4. In this query the set of full matches of the middle leaf is contained in the union of the matches of the left and right leaves.

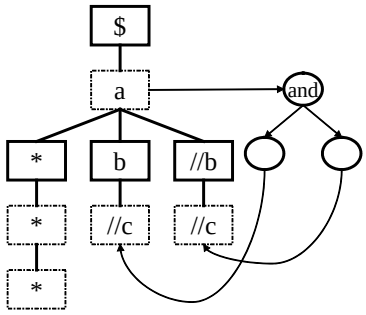

Figure 4: Query tree for `/a[*/*/* and b//c and .//b/c]`

Structural Subsumption-free XPath The set of queries in Subsumption-free XPath that satisfy the stronger requirement that no two of their nodes structurally subsume each other is called *Structural Subsumption-free XPath* (not shown in Figure 3).

Conjunctive XPath Conjunctive XPath is a subset of the mutual intersection of Symmetric XPath, Univariate XPath, and Subsumption-free XPath, which consists of queries that satisfy the two following additional restrictions:

1. All predicates are conjunctions of atomic predicates.
2. None of their wildcard nodes and none of the children of their wildcard nodes have a descendant axis.

5. SPACE LOWER BOUNDS

In this section we prove the space lower bounds on the instance data complexity of XPath evaluation on XML streams. The bounds are stated in terms of three quantitative properties of queries and documents: the *query frontier size*, the *document recursion depth*, and the *document depth*.

The framework for the presentation of each of the lower bounds is as follows. We start with a definition of the XPath fragment $\mathcal{F}$ to which the lower bound is applicable. We then define the quantitative properties of queries and/or documents that are used in the statement of the lower bound. Eventually, we fix an arbitrary query $Q \in \mathcal{F}$ and prove a lower bound on the data complexity of Φ_Q. The bounds are proven w.r.t. streaming algorithms that decide whether a given well-formed XML document matches Q or not. The output of these algorithms on malformed documents can be arbitrary. It follows that the lower bounds hold for stronger types of algorithms as well, including: (1) algorithms that fully evaluate the query on the document and not only decide whether there is a match; (2) algorithms that are designed to evaluate *any* XPath query (not just Q) on any XML document; and (3) algorithms that evaluate the query on well-formed documents and output an error message on malformed documents.

5.1 Query frontier size

Throughout the section, we denote by D_x the subtree of a document D rooted at a node $x \in D$.

We begin with an intuitive overview. Consider any algorithm that evaluates Q on a document D, and suppose $x \in D$ is the node whose start element is currently read from the stream. Let u be a node in Q that x structurally matches. Whether x will turn into a full match of u or not (i.e., whether $\text{EVAL}(u, x) \neq \emptyset$) depends on whether nodes in the subtree D_x (all of which are to appear in later portions of the stream) match the children of u or not. Thus, the algorithm has to allocate space for recording which of the children of u are being matched by nodes in D_x. Moreover, the fate of all the ancestors of u has not be determined yet at the time x is read from the stream. Therefore, the algorithm has to allocate space for recording the status of their children as well. The query frontier of Q at u is the set of u's children and of its ancestors' children. The above discussion implies that the size of the query frontier should be a lower bound on the amount of memory used by the algorithm.

Definition 3. A node y in a rooted tree T is called a *super-sibling* of a node x, if y is either a sibling of x or a sibling of one of its ancestors. The *frontier at* x, denoted $\text{F}(x)$, consists of x and of all of its super-siblings. The *frontier size* of T is $\text{FS}(T) = \max_x |\text{F}(x)|$.

In our running example (Figure 2) the two nodes labeled b and the nodes labeled e and f constitute the frontier at the node labeled e. Since this node is the one with the largest frontier, the size of the frontier of this query is 4.

Before describing the main result of this section, we introduce an intermediate lower bound, on which we build in the proof of the main theorem. This technique allows one to derive memory lower bounds for evaluating a query Q by exhibiting an appropriate "critical" document for Q and considering the frontier size of this document.

The lower bound is applicable to queries in the Symmetric XPath fragment (see Section 4). We first define the notion of critical documents and then state the lower bound:

Definition 4. A document D is called *critical* for a query Q, if D matches Q but if we remove from D any of its subtrees, then the resulting document no longer matches Q.

The document $\langle a\rangle\langle b\rangle 6\langle/b\rangle\langle c\rangle\langle e\rangle\langle/e\rangle\langle f\rangle\langle/f\rangle\langle/c\rangle\langle/a\rangle$ is critical for our example query (Figure 2).

THEOREM 2. *Let Q be any query in Symmetric XPath and let D be any critical document for Q. Then, the space complexity of Φ_Q on XML streams is $\Omega(FS(D))$.*

PROOF. We create from the function Φ_Q a two-argument function Φ_Q^2 as described in Section 3: the first argument is a prefix of an XML stream and the second argument is a suffix of an XML stream.

We use the fooling set technique (see Section 3) to prove the lower bound for Φ_Q^2. We construct a set S of $2^{\text{FS}(D)}$ pairs of the form (α, β), where α and β are, respectively, a prefix and a suffix of an XML stream representing a document that matches Q. In fact, we will choose the pairs such that the documents they form are all identical to the critical document D, except for the order of children in each node.

Let x be the node in D with the largest frontier. We associate with each subset T of $\text{F}(x)$ a pair (α_T, β_T) in S. Thus, $|S| = 2^{|\text{F}(x)|} = 2^{\text{FS}(D)}$, as desired.

For each T, α_T and β_T are SAX event sequences defined as follows. Let $x_1, \ldots, x_\ell$ be the path from the root of D to x (that is, x_1 is the root and $x_\ell = x$). Note that $\text{F}(x)$ consists of x_ℓ and of all the siblings of $x_2, \ldots, x_\ell$. α_T and β_T are formed by concatenating $\ell - 1$ sequences of SAX events: $\alpha_T = \alpha_{T,1} \circ \cdots \circ \alpha_{T,\ell-1}$ and $\beta_T = \beta_{T,\ell-1} \circ \cdots \circ \beta_{T,1}$.

$\alpha_{T,i}$ and $\beta_{T,i}$ are defined as follows. Let $a_i = \text{LABEL}(x_i)$, let $y_1, \ldots, y_k$ be the children of x_i that belong to T, and let $z_1, \ldots, z_m$ the children of x_i that belong to $\text{F}(x)$ but not to T. Then, $\alpha_{T,i}$ is defined as: $\langle a_i\rangle \circ D_{y_1} \circ \cdots \circ D_{y_k}$, where D_{y_j} is the SAX sequence representing the subtree of D rooted at y_j. Similarly, $\beta_{T,i} = D_{z_1} \circ \cdots \circ D_{z_m} \circ \langle/a_i\rangle$.

For the critical document example above the largest frontier consists of the nodes labeled by b, e, f. Consider the set $T = \{b, f\}$. Then,

$$\alpha_T = \langle a\rangle\langle b\rangle 6\langle/b\rangle\langle c\rangle\langle f\rangle\langle/f\rangle$$

and

$$\beta_T = \langle e\rangle\langle/e\rangle\langle/c\rangle\langle/a\rangle.$$

It is easy to verify that $D_T \stackrel{\text{def}}{=} \alpha_T \circ \beta_T$ is identical to D, except that the children of each node x_i on the path from the root to x are ordered as follows: first all the children that belong to T, then x_{i+1}, and then the rest of the children (those that belong to $\text{F}(x) \setminus T$). Since all the predicates in Q are symmetric, $\Phi_Q(D_T) = \Phi_Q(D)$. Therefore, also D_T matches Q.

Consider now two different subsets $T \neq T'$ of $\text{F}(x)$. We show that either $D_{T,T'} \stackrel{\text{def}}{=} \alpha_T \circ \beta_{T'}$ or $D_{T',T} \stackrel{\text{def}}{=} \alpha_{T'} \circ \beta_T$ is a well-formed document that does not match Q. This would imply that S is indeed a fooling set.

It is easy to see that both $D_{T,T'}$ and $D_{T',T}$ are well-formed documents, since the proper nesting of elements is maintained in both. Since $T \neq T'$, then either $T \setminus T' \neq \emptyset$ or $T' \setminus T \neq \emptyset$. Suppose, e.g., that the latter holds. It follows that there is a node y in $\text{F}(x)$, which neither belongs to T nor to $\text{F}(x) \setminus T'$. Let x_i be the parent of y. It is easy to verify from our definition of $\alpha_{T,i}$ and $\beta_{T',i}$ that neither of them contains a copy of D_y. This means that D_y is not contained at all in $D_{T,T'}$. By reordering children of $x_1, \ldots, x_{\ell-1}$ in $D_{T,T'}$, we get an identical copy of D, with the subtree D_y omitted. This document does not match Q, since D is a critical document for Q. It follows that also $D_{T,T'}$ does not match Q (again, using the symmetry of predicates in Q).

We conclude that S is indeed a proper fooling set. The memory lower bound now follows from an application of the fooling set technique (Theorem 1) to the function Φ_Q^2 and by the reduction lemma (Lemma 1). □

The lower bound given by the above theorem may seem non-constructive, because it is not clear how to find a critical document for a given query Q. We address this point in the rest of this section, by showing that for queries in the Conjunctive XPath fragment there exists a critical document whose frontier size is the same as the frontier size of the queries themselves. It then follows that the query frontier size is a lower bound on the memory required to evaluate these queries.

For technical reasons, we assume that all the non-empty atomic predicates in Q cannot be satisfied by string values that start with the special symbol '@'.

THEOREM 3 (MAIN THEOREM). *Let Q be any query in Conjunctive XPath and suppose the alphabet Σ is sufficiently*

large. Then, the space complexity of Φ_Q on XML streams is $\Omega(FS(Q))$.

PROOF. We will construct a critical document D for Q whose frontier size is FS(Q). The lower bound would then follow from Theorem 2.

The function `createCriticalDocument` constructs the critical document D from any query Q in Conjunctive XPath. `getNewSymbol()` is a function that returns a new symbol from Σ, which has not been returned in previous invocations of the function, and which does not occur as a node test in Q. We assume that Σ is large enough to accommodate all the calls to `getNewSymbol()` we make.

Let h denote the length of the longest chain of wildcards in Q; i.e., h is the length of the longest path segment all of whose nodes have the wildcard node test. For example, the longest wildcard chain in the query depicted at Figure 4 is of length 3.

We create two types of nodes in D: "native" and "artificial". For every node $u \in Q$, we create a native node x in D. u is called the "base" of x. If NTEST(u) $\neq *$, then LABEL(x) = NTEST(u). If NTEST(u) $= *$, then LABEL(x) = `getNewSymbol()` (lines 5-8). Let v be the parent of u, and let y be the native node whose base is v. If u has a child axis, then x is set to be a child of y. If u has a descendant axis, then x is set to be a descendant of y, following a chain of $h+1$ new artificial nodes $z_1, \ldots, z_{h+1}$ (lines 1-4). $z_1, \ldots, z_{h+1}$ are assigned labels from `getNewSymbol()`. u is the base of both x and of $z_1, \ldots, z_{h+1}$.

We are left to describe how string values are assigned to the nodes of D. For the given node u, let $u_1, \ldots, u_k$ be the leaf nodes that u structurally subsumes. By our assumption that Q is subsumption-free, SAT(u), the set of satisfying values for u, contains a value α that does not belong to $\bigcup_{i=1}^{k}$ SAT(u_i). In particular this implies that all the leaves $u_1, \ldots, u_k$ have non-empty atomic predicates associated with them. If u is a leaf itself, then we set the value of the corresponding node x to be α. In the pseudo-code we call the function that finds such a value α `getUniqueValue(u)` (lines 9-11). If u is not a leaf, then we set the value of x to be the special symbol '@' (lines 12-13). Using our assumption about non-empty atomic predicates in Q, this implies that the value of x, which is the concatenation of its own value with the values of its descendants, cannot belong to $\bigcup_{i=1}^{k}$ SAT(u_i). Hence, in both cases we are guaranteed that x cannot fully match any of the nodes $u_1, \ldots, u_k$.

Note that the tree representing D is identical to the tree of Q, except that nodes with a descendant axis are expanded to paths of length $h+2$ in D. These paths do not have any effect on the frontier size, since the artificial nodes constituting them do not have any siblings. Hence, the frontier size of D is exactly the same as the frontier size of Q, i.e., FS(Q). To finish the lower bound proof we are left to prove that D is critical for Q.

LEMMA 2. *D is a critical document for Q.*

PROOF. A crucial ingredient of our proof will be the following property of the document D, whose proof is provided below.

LEMMA 3. *Let u be any node in Q and let x be the native node in D whose base is u. Then, x matches u (i.e., EVAL(u, x) $\neq \emptyset$); however, no other node $y \in D$ can match u.*

```
Function createCriticalDocument(Q)
1:   processNode(root(Q))

Function processNode(u)
1:  if (axis(u) = descendant) then
2:    for i := 1 to h + 1 do
3:      t[i] := getNewSymbol()
4:      print '⟨' t[i] '⟩'
5:  a := ntest(u)
6:  if (a = '*')
7:    a := getNewSymbol()
8:  print '⟨' a '⟩'
9:  if (u is a leaf) then
10:   α := getUniqueValue(u)
11:   print α
12: else
13:   print '@'
14:   for c in children(u) do
15:     processNode(c)
16: print '⟨/' a '⟩'
17: if (axis(u) = descendant) then
18:   for i := 1 to h + 1 do
19:     print '⟨/' t[i] '⟩'
```

It follows from Lemma 3 that in particular the root of D matches the root of Q, and therefore D matches Q. We next show how to use Lemma 3 to prove that for any node $x \in D$, the document D_{-x} (D with the subtree rooted at x removed) does not match x. We will assume it does, and reach a contradiction. We will use the following claim:

CLAIM 1. *Let $u \in Q$ be any query node and let E be any document. If a node $x \in E$ matches u, then for each child u' of u there must be a descendant x' of x that matches u'.*

PROOF. Throughout this proof we use the notation S as a shorthand for SMATCH$_{u=x}(u')$.

The predicate associated with u must be satisfied by x, because otherwise EVAL(u, x) $= \emptyset$. If u' is the successor child of u, then EVAL(u, x) $= \bigcup_{x' \in S}$ EVAL(u', x'). Therefore, in order for EVAL(u, x) to be non-empty, there must be some $x' \in S$ (and thus x' is a descendant of x) so that EVAL(u', x') $\neq \emptyset$. Suppose then that u' is a predicate child of u. If EVAL(u', x') $= \emptyset$ for all $x' \in S$, then the atomic predicate that uses the value of u' evaluates to the empty set (= `false`) on x. Since the predicate associated with u is a conjunction of its atomic predicates, also this predicate will evaluate to `false` on x. This is a contradiction to the fact EVAL(u, x) $\neq \emptyset$. □

Since we assumed the document D_{-x} to match Q, then its root matches the root of Q. Let u be the base node of x. By Claim 1 there must be a node $y \in D_{-x}$ that matches u. Since all the predicates in Q are *monotone* (i.e., do not consist of Boolean negations) and symmetric, adding nodes to the document D_{-x} cannot stop y from matching u. This implies that y must match u also in the document D. However, u is the base of x, not of y, and therefore by Lemma 3, y cannot match u in D. We reached a contradiction, and therefore D_{-x} cannot match Q, implying D is a critical document.

PROOF OF LEMMA 3. Proving that x matches its base node u is straightforward from the definition of the document D, and we omit the details from this extended abstract. We next prove that no other node y can match u. To this end, we will need the following lemma, whose proof is provided below.

LEMMA 4. *If a native node $y \in D$ matches a node $u \in Q$, then the base v of y must structurally subsume u.*

We first show how to use Lemma 4 to prove Lemma 3. We prove that y cannot match u by induction on the maximum height of u from leaves of the query tree. The induction base corresponds to nodes u that are leaves themselves. Note that by our construction of the document D only native nodes can match leaves of Q. So assume y is a native node that matches u. By Lemma 4 above, the base of y, v, structurally subsumes u. By our method of assigning values to nodes of D, y is assigned a value which does not belong to any of the sets of satisfying values of leaves that v structurally subsumes. In particular the value of y does not belong to $\text{SAT}(u)$. This implies that y cannot match u.

Assume now that no node y other than x can match a node u of maximum height d. Consider now a node u of maximum height $d+1$. Thus, u has at least one child u' of maximum height d. If y matches u, then by Claim 1, y must have a descendant y' that matches u'. Similarly, since x matches u then x has a descendant x' that matches u'. u' is the base node for x'. If $x' \neq y'$, then by the induction hypothesis y' cannot match u', implying that y does not match u as well. So in order for y to match u, it must be the case that $x' = y'$. This can happen only if u' has a descendant axis, and if either x is an ancestor of y or vice versa.

We first prove that y has to be a native node. If y is an artificial node, then by our method of assigning labels to nodes in D, the only way it can match u is that for u to have a wildcard node test. Recall that u' has a descendant axis and is a child of u. This is impossible because Q belongs to Conjunctive XPath, which forbids wildcard nodes that have children with a descendant axis. We conclude that y cannot be an artificial node. Let v denote the base of y.

Assume, initially, that x is an ancestor of y. Since x' is a descendant of both x and y and y is a descendant of x, then x, y, x' must lie on the same root-to-leaf path in this order. Since all three are native nodes, then the corresponding bases u, v, u' also should lie on the same root-to-leaf path in Q in this order. But this implies that u' is not a child of u, as assumed above. So x cannot be an ancestor of y.

Consider then the case that y is an ancestor of x. By Lemma 4, since y matches u, then v must structurally subsume u. Note, however, that since y is an ancestor of x then v is an ancestor of u. Let k be the depth of v and let k' be the depth of u. The language accepted by $\text{PATHS}(v)$ consists of some strings of length k. However, the language accepted by $\text{PATHS}(u)$ consists only of strings of length at least $k' > k$. Therefore, it cannot be that v structurally subsumes u, implying y cannot be an ancestor of x. We conclude that y cannot match u, as desired. □

PROOF OF LEMMA 4. Let $u_1, \ldots, u_k$ be the path from the root of Q to $u = u_k$ and let $v_1, \ldots, v_\ell$ be the path from the root of Q to $v = v_\ell$. The following proposition, whose proof is omitted, gives necessary and sufficient conditions for v to subsume u:

PROPOSITION 1. *v structurally subsumes u if and only if there exists a mapping $\psi : \{1, \ldots, k\} \to \{1, \ldots, \ell\}$, which satisfies the following conditions:*

1. $\psi(1) = 1$.
2. $\psi(1) < \psi(2) < \cdots < \psi(k)$.
3. *For all $1 \leq i \leq k$, if $\text{NTEST}(u_i) \neq *$, then $\text{NTEST}(v_{\psi(i)}) = \text{NTEST}(u_i)$.*
4. *For all $1 \leq i \leq k$, if u_i has a child axis, then $v_{\psi(i)}$ should also have a child axis.*

For example, a node v whose path is `a/b//c/d` subsumes a node u whose path is `a//c/*`, because there is a mapping ψ that maps location steps of u to location steps of v as follows: $\psi(1) = 1, \psi(2) = 3, \psi(3) = 4$. Note that all the above four properties are maintained.

It would thus suffice to find such a mapping ψ, in order to prove the lemma. Let $y_1, \ldots, y_t$ be the path from the root of D_{-x} to $y = y_t$. The following proposition, whose proof is omitted, gives necessary and sufficient conditions for y to structurally match u:

PROPOSITION 2. *y structurally matches u if and only if there exists a mapping $\phi : \{1, \ldots, k\} \to \{1, \ldots, t\}$, which satisfies the following conditions:*

1. $\phi(1) = 1$.
2. $\phi(1) < \phi(2) < \cdots < \phi(k)$.
3. *For all $1 \leq i \leq k$, if $\text{NTEST}(u_i) \neq *$, then $\text{LABEL}(y_{\phi(i)}) = \text{NTEST}(u_i)$.*
4. *For all $1 \leq i \leq k$, if u_i has a child axis, then $\phi(i) = \phi(i-1) + 1$.*

For example, a document node y whose path is `a/b/c/d` structurally matches a query node u whose path is `a//c/*`, because there is a mapping ϕ that maps location steps of u to labels along the path of y as follows: $\phi(1) = 1, \phi(2) = 3, \phi(3) = 4$. Note that all the above four properties are maintained.

Since y structurally matches u, there exists such a mapping ϕ. We next argue that it is impossible for an artificial node to be in the image of ϕ:

PROPOSITION 3. *For all $1 \leq i \leq k$, $y_{\phi(i)}$ cannot be an artificial node.*

PROOF. Suppose there exists some i, for which y_j, for $j = \phi(i)$, is an artificial node. Since the label of y_j was generated from `getNewSymbol()`, the node test of u_i must be a wildcard. We now prove that if y_j has adjacent artificial nodes they also must belong to the image of ϕ and correspond to wildcard nodes in Q.

Suppose y_{j+1} is an artificial node. If $\phi(i+1) > j+1$, then u_{i+1} has to have a descendant axis. However, we assumed that no child of a wildcard node in Q can have a descendant axis. Therefore, $\phi(i+1) = j+1$. As before, this means that u_{i+1} has a wildcard node test.

Suppose now y_{j-1} is an artificial node. If $\phi(i-1) < j-1$, then u_i has a descendant axis, in contradiction to our assumption that no wildcard node in Q has a descendant axis. Therefore, $\phi(i-1) = j-1$, and as before u_{i-1} has a wildcard node test.

By our construction, any artificial node belongs to a chain of $h+1$ artificial nodes. The argument above implies that if one of these nodes is in the image of ϕ, then all of them must be in the image of ϕ, and they all correspond to nodes whose node test is the wildcard. This means that Q has a chain of wildcards of length $h+1$. This contradicts the fact h is the maximum length of a chain of wildcards in Q.

We conclude that no artificial node can be in the image of ϕ. □

Let $\pi : \{1, \ldots, t\} \to \{1, \ldots, \ell\}$ be the mapping that maps each node on the path leading to y to its base node in Q. We

define the mapping ψ as the composition $\pi \circ \phi$, and prove that it satisfies the four properties specified in Proposition 1.

The first element on the path to v is the root of Q. This root is the base of the root of D, which is the first element on the path to y. Thus, $\pi(1) = 1$. Using the first property of ϕ, we have: $\psi(1) = \pi(\phi(1)) = \pi(1) = 1$.

Next we prove ψ is monotone increasing. That is, for each $i = 1, \ldots, k-1$, $\psi(i) < \psi(i+1)$. Since ϕ is monotone increasing and π is monotone non-decreasing, we know that $\psi(i) \leq \psi(i+1)$. Thus, we just need to exclude the possibility that $\psi(i) = \psi(i+1)$. Suppose $\psi(i) = \psi(i+1)$; thus, $\pi(\phi(i)) = \pi(\phi(i+1))$, even though $\phi(i) < \phi(i+1)$. The only way two nodes have the same base is that the first of them is artificial. Thus, $y_{\phi(i)}$ has to be an artificial node. However, by Proposition 3, no artificial node can be in the image of ϕ. Therefore, we must have $\psi(i) < \psi(i+1)$.

For the third property, let $1 \leq i \leq k$ be s.t. $\text{NTEST}(u_i) \neq *$. By the third property of ϕ, the label of $y_{\phi(i)}$ equals $\text{NTEST}(u_i)$. This label is a node test that occurs in Q; hence, the label of $y_{\phi(i)}$ could not have been generated from `getNewSymbol()`. Therefore, the node test of $v_{\psi(i)}$, the base node of $y_{\phi(i)}$, cannot be a wildcard. We have now from our construction, $\text{NTEST}(v_{\psi(i)}) = \text{LABEL}(y_{\phi(i)}) = \text{NTEST}(u_i)$.

Finally, for the fourth property, let $1 \leq i \leq k$ be such that u_i has a child axis. By the fourth property of ϕ, $\phi(i-1) = \phi(i) - 1$. By Proposition 3 both $y_{\phi(i)}$ and $y_{\phi(i-1)}$ cannot be artificial nodes. Let $v_{\psi(i)}$ and $v_{\psi(i-1)}$ be their corresponding base nodes. If $v_{\psi(i)}$ had a descendant axis, then $y_{\phi(i)}$ would have been separated from $y_{\phi(i-1)}$ by a chain of $h+1$ artificial nodes. However, we know that $y_{\phi(i)}$ and $y_{\phi(i-1)}$ are adjacent. Therefore, $v_{\psi(i)}$ mush have a child axis.

We conclude that ψ satisfies the four properties of Proposition 1, and therefore v indeed structurally subsumes u.

5.2 Recursion depth

The *recursion depth* of a document D with respect to a node v in a query Q is the length of the longest sequence of nodes $x_1, \ldots, x_r \in D$, such that: (1) all of them lie on the same root-to-leaf path; and (2) all of them structurally match v. For example, if Q is `//a[b and c]` and D is $\langle a\rangle\langle a\rangle\langle b\rangle\langle/b\rangle\langle c\rangle\langle/c\rangle\langle/a\rangle\langle/a\rangle$, then the recursion depth of D w.r.t. the node labeled `a` is 2.

We next show that for queries Q that include the expression `//a[b and c]`, the document recursion depth is a lower bound on the space complexity of Φ_Q in the data stream model. In order to make the argument precise, we need to make sure this expression is a "vital" part of the query Q. To this end, as in the previous lower bound, we need to restrict Q to the Conjunctive XPath fragment (see Section 4). Thus, the queries Q to which the lower bound below applies are those in Conjunctive XPath that have at least one node v that satisfies the following requirements: (1) Either v or an ancestor of v have a descendant axis. (2) v has at least two children with a child axis. We denote by $\mathcal{F}$ the fragment of Conjunctive XPath that satisfies the above requirement.

THEOREM 4. *Let Q be any query in $\mathcal{F}$. Let v be the node of Q, as defined above. Then, the space complexity of Φ_Q on XML streams is $\Omega(r)$, where r is the recursion depth of the document w.r.t. v.*

PROOF. We use a reduction from the set disjointness problem in communication complexity. In set disjointness, DISJ, Alice and Bob get Boolean vectors $\mathbf{s}, \mathbf{t} \in \{0,1\}^r$, respectively. $\mathbf{s}$ and $\mathbf{t}$ are viewed as characteristic vectors of two sets $S, T \subseteq \{1, \ldots, r\}$ (that is, $\mathbf{s}_i = 1$ if and only if $i \in S$, and similarly for $\mathbf{t}$ and T). $\text{DISJ}(\mathbf{s}, \mathbf{t}) = 1$ if and only if $S \cap T \neq \emptyset$. The communication complexity of DISJ is $\Omega(r)$ (cf. [18]).

We will prove that given a streaming algorithm for evaluating Q on documents of recursion depth r w.r.t. v using space S, we can design a protocol that solves the set disjointness problem with S bits of communication. It would then immediately follow that S has to be at least $\Omega(r)$.

Let u be the lowest ancestor of v that has a descendant axis (if v itself has a descendant axis, then $u = v$). Let w, w' be two children of v that have a child axis.

In the reduction we use the same construction of a critical document D for Q as the one used in the proof of Theorem 3. Recall that in that construction each node q of Q has a corresponding "native" node d in D, and q is called the *base* of d. Let then x be the native node whose base is u, y the native node whose base is v, and z, z' the native nodes whose bases are w, w'. Let a, b, c, c' be the labels of x, y, z, z', respectively. Consider the stream representation of D, and split it into 9 segments as follows:

$$\phi_1\langle a\rangle\phi_2\langle b\rangle\phi_3\langle c\rangle\phi_4\langle/c\rangle\phi_5\langle c'\rangle\phi_6\langle/c'\rangle\phi_7\langle/b\rangle\phi_8\langle/a\rangle\phi_9.$$

That is, the segments that come before, after, and inside each of the nodes x, y, z, z'.

We are now ready to describe the protocol for set disjointness. Given her input $\mathbf{s}$, Alice prepares a prefix of an XML stream α that starts with ϕ_1 and continues with r consecutive segments $\alpha_1, \ldots, \alpha_r$. $\alpha_i = \langle a\rangle\phi_2\langle b\rangle\phi_3\langle c\rangle\phi_4\langle/c\rangle\phi_5$, if $\mathbf{s}_i = 1$, and $\alpha_i = \langle a\rangle\phi_2\langle b\rangle\phi_3\phi_5$, if $\mathbf{s}_i = 0$. That is, α_i includes a copy of the node z, only if $\mathbf{s}_i = 1$. Similarly, given his input $\mathbf{t}$, Bob prepares a suffix of an XML stream β that starts with r consecutive segments $\beta_r, \ldots, \beta_1$, and ends with ϕ_9. $\beta_i = \langle c'\rangle\phi_6\langle/c'\rangle\phi_7\langle/b\rangle\phi_8\langle/a\rangle$, if $\mathbf{t}_i = 1$, and $\beta_i = \phi_7\langle/b\rangle\phi_8\langle/a\rangle$, if $\mathbf{t}_i = 0$. That is, β_i includes a copy of the node z', only if $\mathbf{t}_i = 1$.

The document $D_{\alpha,\beta}$ represented by the concatenation $\alpha \circ \beta$ is well formed, because nesting of elements is properly maintained. The document has recursion depth r w.r.t. v, because it contains r instances of the node y nested within each other and all of them structurally match v. The following lemma, whose proof is omitted from this extended abstract, gives us a necessary and sufficient condition for $D_{\alpha,\beta}$ to match Q:

LEMMA 5. *$D_{\alpha,\beta}$ matches Q if and only if there exists some index $1 \leq i \leq r$, s.t. $\mathbf{s}_i = \mathbf{t}_i = 1$.*

The protocol for set disjointness proceeds as follows. Alice runs the given streaming algorithm (that evaluates Q) on the XML stream prefix α. When she is done, she sends the state of the algorithm (which consists of at most S bits) to Bob. Bob can continue the execution of the algorithm on the suffix β. At the end of the execution, if the algorithm decides that there is a match, Bob declares the sets S and T to be intersecting. Otherwise, he declares them to be disjoint.

First, it is obvious that this protocol indeed uses only S bits of communication. We next prove that it computes the function DISJ correctly. Suppose, initially, that $S \cap T \neq \emptyset$. Therefore, there exists some index $1 \leq i \leq r$, such that both $\mathbf{s}_i$ and $\mathbf{t}_i$ are 1. By Lemma 5, this means that the

document $D_{\alpha,\beta}$ matches Q, and therefore the algorithm will find a match. Hence, the protocol will indeed declare **s** and **t** as intersecting. The analysis of the case $S \cap T = \emptyset$ is similar. □

5.3 Document depth

The *depth* of a document is the length of the longest root-to-leaf path in the tree representing the document. We prove the following lower bound in terms of the document depth:

THEOREM 5. *Let Q be any query in Conjunctive XPath that has at least one node u s.t. (1) u has a child axis; (2) $\text{NTEST}(u) \neq *$. Then, the space complexity of Φ_Q on XML streams is $\Omega(\log d)$, where d is the document depth.*

PROOF. We create from Φ_Q a two-argument function Φ_Q^3 (recall our notations from Section 3): its first argument is pair (α, γ), where α is a prefix of an XML stream and γ is a suffix of an XML stream; its second argument β is the middle part of an XML stream.

We use the fooling set technique from Section 3. We thus need to create a set S of $t = O(d)$ documents $D_1, \ldots, D_t$ of depth at most d that match Q. We then split each document D_i into three parts: α_i, β_i, and γ_i, and show that for all $i \neq j$, one of the documents $\alpha_i \circ \beta_j \circ \gamma_i$, $\alpha_j \circ \beta_i \circ \gamma_j$ is well-formed but does not match Q.

Let D be a critical document for Q produced according to the construction described in the proof of Theorem 3, and let x be the node in D whose base is u. Let g be some symbol in the alphabet that does not occur as a label in D. We set $t = d - s$, where s is the depth of the document D. $D_1, \ldots, D_t$ will be all created from D as follows. D_i is the same as D, except that we attach to it two paths of length i, all of whose nodes are labeled by the symbol g. The first node of each of these two paths is attached as a sibling of x: the first node of the first path is attached just before x, and the first node of the second path is attached just after x.

We split the SAX sequence representing D_i into three non-overlapping parts: α_i consists of all the events until after the start-element event corresponding to the last node on the first new path we attached; β_i consists of all the events until after the start-element event corresponding to the last node on the second new path we attached; γ_i consists of the remainder of the sequence. For example, if the query Q is `a/b`, then $\alpha_i = \langle a\rangle\langle g\rangle^i$, $\beta_i = \langle/g\rangle^i\langle b\rangle\langle/b\rangle\langle g\rangle^i$, and $\gamma_i = \langle/g\rangle^i\langle/a\rangle$.

First, since all the predicates in Q are monotone, then whether D matches Q or not does cannot change by addition of new siblings to x. Therefore, $D_1, \ldots, D_t$ all match Q. We next prove that for $i > j$, $D_{i,j} \stackrel{\text{def}}{=} \alpha_i \circ \beta_j \circ \gamma_i$ is a well-formed document that does not match Q.

The easiest way to see what happens in the document $D_{i,j}$ is to consider the example `a/b`. For this query, $D_{i,j} = \langle a\rangle\langle g\rangle^i\langle/g\rangle^j\langle b\rangle\langle/b\rangle\langle g\rangle^j\langle/g\rangle^i\langle/a\rangle$. That is, the node x (labeled in this example by the symbol `b`) becomes the child of the $(i-j)$-th node on the first new path we inserted. Note that the proper nesting of elements is maintained, and therefore $D_{i,j}$ is a well-formed document. Also in the general case $D_{i,j}$ is a well-formed document. The following lemma, whose proof is omitted from this extended abstract, uses the special properties of the critical document D to show that $D_{i,j}$ cannot match Q:

LEMMA 6. *$D_{i,j}$ does not match Q.*

We conclude that S is indeed a fooling set of size $t = d - s$. Applying Theorem 1 to the function Φ_Q^3 and Lemma 1 give us a space lower bound of $\log(d-s)/2 = \Omega(\log d)$. □

6. UPPER BOUNDS

In this section we describe an XPath filtering algorithm, whose space is close to the lower bounds described in the previous section. The algorithm handles any query in Univariate XPath. An example of its use is provided in Section 6.1.

We have six global data structures: (1) pointer array: array of query nodes used to identify the currently "matched" query nodes, i.e., the query nodes that we are currently processing; (2) validation array: Boolean array used for checking if a given query node has already been satisfied; (3) level array: integer array used to identify the document level for each matched node; (4) recursion level array: integer array that keeps track of the recursion level for each query node; (5) next index: integer variable that holds the index of the next entry to be added to the pointer, validation, and level arrays; (6) current level: integer that holds the level of the currently processed document node.

The algorithm works by processing the `startElement` and `endElement` events of the SAX interface. In addition, it uses two auxiliary functions: `initialize` and `evalPred`. Function `initialize` is executed only once, at the beginning of processing. It initializes the global variables. In particular, it sets the pointer array to match the query "root" (line 1), the validation array for the root to `false` (line 2), indicating that the root is not satisfied yet, and the level array for the root to 0 (line 3), indicating that a match for the root should happen at level 0. The pointer, validation, and level arrays always have the same size and are initialized at runtime as they are incremented. The recursion level array has one entry per query node, and it is initialized to 0 (lines 4-5), since during initialization no recursive matches have yet happened. Finally, the current level and next index variables are initialized to indicate that the current document level is 0 and that there exists one node in the pointer array, which is the root (lines 6-7).

Function `startElement` is called once for every open element event in the document stream. It first checks if the document node opened matches one of the query nodes we are currently processing (lines 2-6). A match occurs if the document node label matches the query node test name or if the query node is * (line 4). In addition, if the query node has a child axis the levels must be same. The level is ignored for query nodes with descendant axis (lines 5-6). In the case of a match, we first check if the current node has already been satisfied by checking its validation array entry. `startElement` only processes nodes that have not been satisfied yet (line 7). This enforces the existential semantics of XPath. The children of nodes that still need to be processed are added to the pointer and level arrays, since we are now trying to match these nodes (lines 8-17). If the node being added to the arrays is the first child of u and if there are no other instances of that node in the array (i.e. recursion level for u is 0) it "reuses" u's position in the arrays (lines 9-12). All the other children of u are added to the end of the array (lines 13-17). Finally, the recursion level for the node u and the current document level are incremented (lines 18-19).

Function `endElement` is called once for every close element event in the document stream. It starts by decrementing the

```
Function initialize()
1:  pointerArray[0] := $
2:  validationArray[0] := false
3:  levelArray[0] := 0
4:  for i:= 0 to (|Q| - 1)
5:    recursionArray[i] = 0
6:  currentLevel := 0
7:  nextIndex := 1

Function startElement(x)
1:  maxIndex := nextIndex - 1
2:  for i := 0 to maxIndex do
3:    u := pointerArray[i]
4:    if ((ntest(u) = label(x)) or (ntest(u) = *)) then
5:      if ((axis(u) = descendant) or
6:          (levelArray[i] = currentLevel)) then
7:        if (not validationArray[i]) then
8:          for c in children(u) do
9:            if ((c = firstSibling(c)) and
10:               (recursionArray[u] = 0)) then
11:             pointerArray[i] := c
12:             levelArray[i] := currentLevel + 1
13:           else
14:             pointerArray[nextIndex] := c
15:             levelArray[nextIndex] := currentLevel + 1
16:             validationArray[nextIndex] := 0
17:             nextIndex := nextIndex + 1
18:         recursionArray[u] := recursionArray[u] + 1
19: currentLevel := currentLevel + 1
```

```
Function endElement(x)
1: currentLevel := currentLevel - 1
2:  i := nextIndex - 1
3:  while (levelArray[i] > currentLevel) do
4:    u := pointerArray[i]
5:    if ((u = firstSibling(u) and
6:        (recursionArray[parent(u)] = 0)) then
7:      pointerArray[i] := parent(u)
8:      levelArray[i] := currentLevel
9:    else
10:     nextIndex := nextIndex - 1
11:   i := i - 1
12: for i := 0 to (nextIndex - 1) do
13:   u := pointerArray[i]
14:   if ((ntest(u) = label(x)) or (ntest(u) = *)) then
15:     if ((axis(u) = descendant) or
16:         (levelArray[i] = currentLevel)) then
17:       if (isLeaf(u)) then
18:         validationArray[i] := evalPred(u)
19:       else
20:         c := validationArray bits for children(u)
21:         if (evalPred(c)) then validationArray[i] := true
22:       recursionArray[u] := recursionArray[u] - 1
23:       if (u = $) then
24:         queryResponse := validationArray[i]
```

current level (line 1). It then checks if there are nodes in the pointer array that need to be removed since their parent is the node being closed (lines 2-10). If the node being removed from the arrays is the first child of its parent and if the recursion level for the parent is 0, the parent node is added back to the array (lines 5-8). On the other hand, if the node is removed from the end of the arrays, *nextIndex* is updated (lines 9-10). `endElement` then updates the validation array entries for the nodes being closed (lines 12-22). If the node being closed is a leaf, the validation array is set by evaluating the predicate on that node only (lines 17-18). In the case of intermediate nodes the predicate is evaluated by checking the validation array entries of all their children (lines 20-21). 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 the absence of predicates `evalPred` returns `true` for all leaf nodes and returns an AND over the validation array entries of the children of intermediate nodes. (We omit a procedural definition of `evalPred` due to lack of space.) After updating the validation array entry for the matched node, `endElement` checks if that node is the "root". In that case we have the response for the query, which is the value of its validation array entry (lines 23-24).

Remark: Since in this paper we concentrate on the evaluation of queries in Univariate XPath, buffering is not required for evaluating predicates. In [17] we provide an in-depth discussion of predicate evaluation in Forward XPath that may require buffering.

THEOREM 6. *For queries in Univariate XPath, the space complexity of the above algorithm is* $O(|Q| \cdot r \cdot (\log |Q| + \log d + \log r))$, *where* $|Q|$ *is the query size,* r *is the document recursion depth, and* d *is the document depth. For queries in Structural Subsumption-free XPath and non-recursive documents, the space complexity is* $O(FS \cdot (\log |Q| + \log d))$, *where* FS *is the query frontier size.*

PROOF. A query node u is inserted into the pointer array, only if its parent v is structurally matched by some node in the document. The pointer array (and therefore also the validation and level arrays) can have multiple copies of u simultaneously, only if several document nodes structurally match v and all of them are nested within each other. Therefore, the maximum number of entries in these arrays for queries in Univariate XPath is $|Q| \cdot r$. The recursion level array has the fixed size of $|Q|$, since it has one entry per query node. Thus, the space complexity is $O(|Q| \cdot r \cdot (\log |Q| + \log d + \log r))$. For queries in Structural Subsumption-free XPath our algorithm guarantees that, at any given moment, the set of nodes contained in the pointer array form a frontier. Therefore, for this type of queries and for non-recursive documents, the space complexity goes down to $O(FS \cdot (\log |Q| + \log d))$. This matches the lower bound, modulo logarithmic factors. □

THEOREM 7. *For queries in Univariate XPath, the time complexity of the above algorithm is* $O(|D| \cdot |Q| \cdot r)$, *where* $|D|$ *is the document size. For queries in Structural Subsumption-free XPath and for non-recursive documents, the time complexity is* $O(|D| \cdot FS)$.

PROOF. The query nodes accessed in the `startElement` and `endElement` functions are those that are currently contained in the pointer array and possibly their children. The number of such nodes is at most $|Q| \cdot r$ for queries in Univariate XPath and FS for queries in Structural Subsumption free XPath and for non-recursive documents. Since the above two functions are called for each document node, the total running time is at most $O(|D| \cdot |Q| \cdot r)$ for queries in Univariate XPath and $O(|D| \cdot FS)$ for queries in Structural Subsumption-free XPath and for non-recursive documents. □

6.1 Example run

In Figure 5 we present an example of how the algorithm processes the query `a[c[.//e and f] and b]`, which is a simplified version of our running example. We show a sample document and a snapshot of the state of the main data structures after each event. We use tuples to represent the values for the validation, pointer, and level arrays for each array entry. Since the document is not recursive, we omit the recursion level array. Each event is represented by the tag name and the level it happened. Index 0, 1, and 2 represent indices into the arrays. The first interesting event is

the open d (event 4). Since d is not matched we increase the level by one but keep the arrays intact. The other interesting event is the second open c (event 11). Since c is already matched at that point, instead of processing it again we simply increment the level variable. This makes sure the existential semantics of XPath is preserved. Note that the second c would not match the query and its evaluation would incorrectly set the second bit of the validation array to 0 (`false`). At the end of processing we simply check index 0 of the validation array, which is 1 (`true`) meaning that the document matched the query.

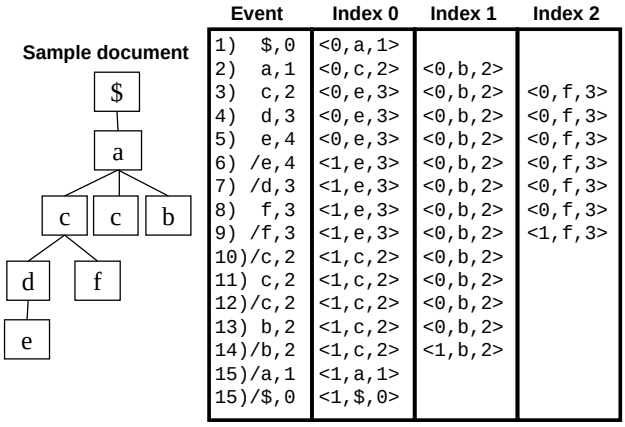

Figure 5: Example run for `/a[c[.//e and f] and b]`

7. CONCLUSIONS

In this paper we present the first systematic and theoretical study of memory lower bounds for XPath queries over XML streams. We presented the minimum amount of memory that *any* algorithm evaluating the query on a stream would need to incur. We also presented a new XPath filtering algorithm that uses space close to the optimum. To the best of our knowledge this algorithm has the best theoretical efficiency guarantees among all known streaming algorithms for XPath filtering, both in terms of memory consumption and running time. Our future research directions include extending our results to larger classes of queries, such as ones with multi-variable predicates that require buffering.

Acknowledgments

We would like to thank Ron Fagin, Moshe Vardi, and the anonymous referees for helpful comments.

8. REFERENCES

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

[2] 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.

[3] 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.

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

[5] T. Bray, J. Paoli, and C. M. Sperbeg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C, `http://www.w3.org/TR/1998/REC-xml-19980210`, 1998.

[6] 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.

[7] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *Proc. 14th DEXA*, pages 28–37, 2003.

[8] B. Choi, M. Mahoui, and D. Wood. The optimality of holistic algorithms for XPath. `http://www.cis.upenn.edu/~kkchoi/xpath.pdf`, 2003.

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

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

[11] 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.

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

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

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

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

[19] 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.

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

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

[22] M. Y. Vardi. The complexity of relational query languages. In *Proc. 14th STOC*, pages 137–146, 1982.

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