On the Memory Requirements of XPath Evaluation over XML Streams[*]

Ziv Bar-Yossef[*] Marcus Fontoura[†] Vanja Josifovski[†]

Abstract

The important challenge of evaluating XPath queries over XML streams has sparked much interest in the past few 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 proofs are 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 [8] is quickly gaining dominance as a format for exchanging and storing semi-structured data. The most popular language for querying XML data is XPath [13, 6], which is part of both XSLT [12] and XQuery [7], 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, 10, 14, 18, 20, 21, 22, 25, 26]. 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 [18, 26], 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 [29]. As opposed to previous results in the area [18], 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.

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 [22].

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* FULLEVAL which maps query-database pairs (Q, D) into output values. By fixing a query Q, we get an induced mapping FULLEVAL$_Q$ from databases to output values. Similarly, by fixing a database D, we obtain an induced mapping FULLEVAL$_D$ from queries to output values. Vardi [28] defined three standard measures of complexity for database query languages: the *data complexity* (the complexity of

2

FULLEVAL$_Q$, for the worst-case choice of Q), the *expression complexity* (the complexity of FULLEVAL$_D$, for the worst-case choice of D), and the *combined complexity* (the complexity of FULLEVAL). 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* [15]. 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 $\{$FULLEVAL$_Q\}_{Q \in \mathcal{F}}$, where $\mathcal{F}$ is a large fragment of the query language. Naturally, for different queries, the corresponding mappings may have different complexities. Thus, the complexity of FULLEVAL$_Q$ 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 the mappings FULLEVAL$_Q$, for all the queries Q in the query language 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 FULLEVAL$_Q$ for all $Q \in \mathcal{F}$.

Our results Empirically, the bulk of memory used by algorithms that evaluate XPath queries over XML streams 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. In this paper we prove three memory lower bounds that address both sources of memory consumption.

Our first 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 FULLEVAL$_Q$ 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 [18, 20, 25, 26]. 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 non-deterministic

automata by deterministic ones.[1] 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. The lower bound is not applicable to arbitrary queries, but rather only to a large fragment of XPath that we define. This fragment, called "Redundancy-free XPath", consists of queries that satisfy certain restrictions. The most important of these is that queries should not have redundant parts that can be eliminated without changing the semantics of the queries.

Our two other lower bounds relate to the use of memory for buffering (representations of) document fragments. 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 queries in a large subset of Redundancy-free XPath 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 queries in a large subset of Redundancy-free XPath 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 is $\tilde{O}(|Q| \cdot r \cdot \log d)$, where $|Q|$ is the query size, r is the document's recursion depth, d is the document depth, and $\tilde{O}$ suppresses logarithmic factors. Thus, the algorithm separately (almost) matches the recursion depth and document depth lower bounds. For a certain class of queries, when applied to non-recursive documents, the algorithm uses $\tilde{O}(\mathrm{FS}(Q) \cdot \log d)$ bits of space, where $\mathrm{FS}(Q)$ is the frontier size of Q. Thus, for these queries the algorithm almost matches also the query frontier size lower bound.

The proposed algorithm builds on our recent XQuery evaluation algorithm for XML streams [22], 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 closer to the query frontier size lower bound.

The rest of the paper is organized as follows. Section 2 overviews related work. In

---

[1]Although queries are usually assumed to be small relative to the database size, exponential space in the size of the query may be prohibitive, even for queries that consist of as few as 30 nodes.

Section 3 we provide background material on XML and XPath, streaming algorithms, and communication complexity. In Section 4 we prove our three lower bound for three specific, carefully chosen, queries. The goal is to give the reader a gentle introduction to the proof techniques, before delving into the intricacies of Redundancy-free XPath. Section 5 defines Redundancy-free XPath. Section 6 provides some of the technical machinery used throughout our proofs to argue about matchings of documents with queries. In Section 7 we describe and prove the three lower bounds in their most general form. In Section 8 we outline the new filtering algorithm. We conclude in Section 9 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, 10, 14, 18, 20, 21, 22, 25, 26]. 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 $\tilde{O}(|Q| \cdot r \cdot \log d)$ space and $\tilde{O}(|Q| \cdot |D| \cdot r)$ time.

Gottlob, Koch, and Pichler [17] and Segoufin [27] 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 [11] consider memory lower 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 FULLEVAL$_Q$ has constant ("bounded") space complexity

and ones that have unbounded space complexity. We give a finer estimation of the space complexity of FULLEVAL$_Q$, for all Q.

Grohe, Koch, and Schweikardt [19] consider streaming algorithms that allow multiple sequential scans of the data. They show tradeoffs between the space and the number of scans needed for evaluation of certain XPath queries. While the model they consider is more general than ours, they prove lower bounds on the worst-case data complexity, while our bounds hold for the instance data complexity.

In subsequent work [5], we extended the methodology of this paper to address other sources of buffering. In particular, we showed that full-fledged evaluation of queries (as opposed to just filtering) and evaluation of queries with multi-variable predicates (as opposed to single-variable predicates) require large buffers. We also prove that together with recursion, which is discussed in this paper, these exhaust the factors that necessitate buffering in XPath evaluation over XML streams.

3   Preliminaries

Notations   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, we will denote its root by ROOT(T). For a node $x \in T$, we denote by T_x the subtree of T rooted at x. For two nodes $x, y \in T$, where x is an ancestor of y, we denote by PATH($x..y$) the sequence of nodes along the path from x to y (inclusive). PATH(x) is simply the sequence PATH(ROOT(T)$..x$).

$\mathcal{N}$ 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 two strings or sequences, α and β, $\alpha \circ \beta$ denotes the string/sequence obtained by concatenating α and β.

3.1   XPath

Data model   We use the XPath 2.0 and XQuery 1.0 Data Model [16]. An XML document is a rooted tree. Every node x has the following properties:

1. 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. NAME(x), which is a value from $\mathcal{N}$. `root` and `text` nodes are unnamed.

3. STRVAL(x), which is a string from $\mathcal{S}$. 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. DATAVAL(x), which is a data value from $\mathcal{V}$. DATAVAL(x) is derived from STRVAL(x), using the document's XML schema.

XPath Figure 1 describes *Forward XPath*—a fragment of XPath 2.0 [6], which supports only the forward axes. The main XPath fragment considered in this paper is a subset of Forward XPath (see Section 5).

```
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 $\in$ { =, !=, <, <=, >, >= }.
arithop $\in$ { +, -, *, div, idiv, mod }.
funcop is any basic XPath function or operator on atomic arguments as specified in [24], 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. AXIS(u), which in this paper can be either child, attribute, or descendant.[2] 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]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.

2. NTEST(u), which is either a name from $\mathcal{N}$ or the wildcard $*$. The root does not have a node test.

3. SUCCESSOR(u), which is either empty or one of the children of u.

4. PREDICATE(u), which is either empty or an expression tree, as described below.

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 LEAF(u). The succession leaf of the root is called the *query output node*, and is denoted by 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.

Example. Figure 2 shows an example query tree. The successor of a node is marked by a dashed box. For example, the successor of the root node is the node named "a" and, in turn, the successor of the node named "a" is the second node (going from left to right) named "b". Each node is annotated by an XPath axis. We use / to indicate `child` axis and // for `descendant` axis. The root is annotated with the "$" sign. The predicate of a given node is represented by a "predicate tree" pointed to by that node. In this example the nodes named "a" and "c" have predicates. In Figure 2, the node named "c" and the first node named "b" are predicate children of the node named "a", and the nodes named "e","f" are predicate children of the node named "c".

Query evaluation   For the rest of the section, fix Q and D to be some arbitrary Forward XPath query and XML document, respectively. The evaluation of Q on D specifies a sequence of nodes that Q "selects" from D, in document order. A formal definition of this function appears below.

Definition 3.1 (Node test passage). A name $n \in \mathcal{N}$ is said to *pass a node test N*, if either $N = n$ or $N = *$.
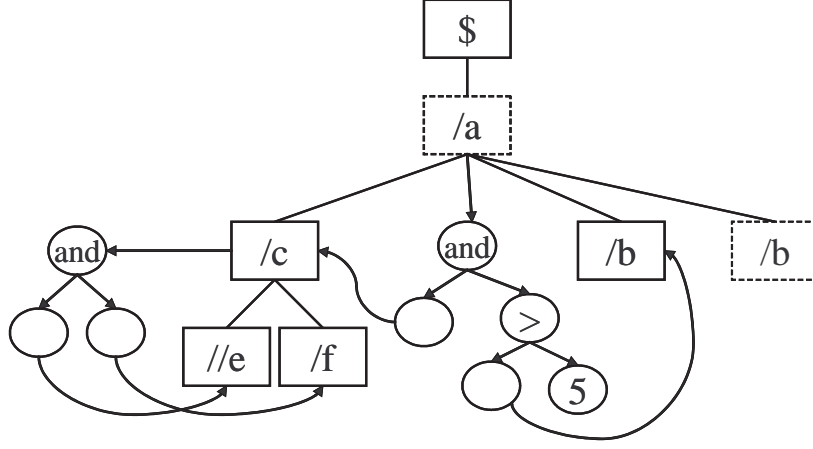
Figure 2: Query tree for `/a[c[.//e and f] and b > 5]/b`

Definition 3.2 (Axis-specified tree-relationship). Let x, y be two nodes in a rooted tree T, and let A be an axis. y is said to *relate to x according to the axis A*, if one of the following holds:

1. $A = \texttt{child}$ and y is a child of x.

2. $A = \texttt{descendant}$ and y is a descendant of x.

Definition 3.3 (Predicate satisfaction). Let u be any query node and let $\text{PREDICATE}(u)$ be the predicate of u. A document node x is said to *satisfy the predicate $\text{PREDICATE}(u)$*, if one of the following holds:

1. $\text{PREDICATE}(u)$ is empty.

2. $\text{PREDICATE}(u)$ is not empty, and $\texttt{EBV}(\text{PEVAL}(r_u, x)) = \texttt{true}$, where r_u is the root of $\text{PREDICATE}(u)$, $\text{PEVAL}(\cdot, \cdot)$ is the predicate evaluation function defined below, and $\texttt{EBV}(\cdot)$ is the Effective Boolean Value function discussed below.

Definition 3.4 (Node selection). Let v be any query node, which is not the root. Let u be any node in $\text{PATH}(v)$, and let x be any document node of kind $\texttt{root}$ or $\texttt{element}$. The *node sequence selected by the node v under the context $u = x$*, denoted $\text{SELECT}(v|u = x)$, is defined inductively as follows.

- If $u = v$, then $\text{SELECT}(u|u = x) = \{x\}$.

- If $u = \text{PARENT}(v)$, then $\text{SELECT}(v|\text{PARENT}(v) = x)$ consists of the sequence of document nodes y that satisfy the three following conditions, in document order:

9

1. $\text{NAME}(y)$ passes $\text{NTEST}(v)$.

2. y relates to x according to $\text{AXIS}(v)$.

3. y satisfies $\text{PREDICATE}(v)$.

- If $u \neq \text{PARENT}(v)$, then u is an ancestor of $\text{PARENT}(v)$. By induction, let $(z_1, \ldots, z_n)$ the sequence selected by $\text{PARENT}(v)$ under the context $u = x$. We now define:

$$\text{SELECT}(v|u = x) = \text{SELECT}(v|\text{PARENT}(v) = z_1) \circ \cdots \circ \text{SELECT}(v|\text{PARENT}(v) = z_n).$$

Definition 3.5 (Predicate evaluation). Let u be a query node, let $\text{PREDICATE}(u)$ be the predicate of u, and let x be a document node. The *evaluation of a node $s \in \text{PREDICATE}(u)$ on x*, denoted $\text{PEVAL}(s, x)$, is either an atomic value or a sequence and is defined recursively as follows.

1. If s is labeled by a constant $c \in \mathcal{V}$, then $\text{PEVAL}(s, x) = c$.

2. If s is labeled by a pointer to a child v of u, let $\text{LEAF}(v)$ be the succession leaf of v. Then, $\text{PEVAL}(s, x)$ is the sequence of data values of the nodes in $\text{SELECT}(\text{LEAF}(v)|u = x)$.

3. If s is labeled by a function or operator f whose arguments are boolean (e.g., the logical operators `and,or,not`) and the children of s are $t_1, \ldots, t_k$, then

$$\text{PEVAL}(s, x) = f(\text{PEVAL}(t_1, x), \ldots, \text{PEVAL}(t_k, x)),$$

where the arguments to f are first cast to boolean by the `EBV` function (see below).

4. If s is labeled by an operator or a function f whose output is boolean but whose arguments are non-boolean (e.g., comparison operators) and the children of s are $t_1, \ldots, t_k$, then

$$\text{PEVAL}(s, x) = \texttt{true} \iff \exists \alpha_1 \in P_1, \ldots, \exists \alpha_k \in P_k \text{ s.t. } f(\alpha_1, \ldots, \alpha_k) = \texttt{true}.$$

Here, for each $i = 1, \ldots, k$, P_i is a sequence defined as follows. If $\text{PEVAL}(t_i, x)$ is an atomic value, then P_i is a length 1 sequence, consisting of this value, after proper conversion to the type required by f. If $\text{PEVAL}(t_i, x)$ is a sequence, then P_i is the same sequence, after proper conversion of each element to the type required by f.

5. If s is labeled by an operator or a function f whose arguments are non-boolean and whose output is non-boolean (e.g., arithmetic operators) and the children of s are $t_1, \ldots, t_k$, then

$$\text{PEVAL}(s, x) = (f(\alpha_1, \ldots, \alpha_k) \ : \ \alpha_1 \in P_1, \ldots, \alpha_k \in P_k).$$

10

Here, $P_1, \ldots, P_k$ are sequences defined as above. The sequence $\text{PEVAL}(s, x)$ is formed by going over all $\alpha_1 \in P_1, \ldots, \alpha_k \in P_k$ in lexicographical order.

In the above evaluations standard conversions among the various XPath types are applied. The most important conversion rule, is defined by the *Effective Boolean Value* (`EBV`) function. This function converts a data value of any type into a boolean value. `EBV` is used to cast the output of the predicate's root into a boolean, as well as to cast the operands of boolean operators (e.g., `and,or,not`) into boolean. When the operand of `EBV` is a sequence, it returns true if the sequence is not empty, giving most XPath expressions an existential semantics.

Remark. Definition 3.5 slightly deviates from the standard specifications of XPath [6]. The existential evaluation rule (part 4) applies in the standard specification only to comparison operators (i.e., `=, !=, <, <=, >, >=`), and not to every function whose output is boolean. Furthermore, if a node s is labeled by an operator or a function f on non-boolean arguments (excluding the comparison operators), then $\text{PEVAL}(s, x)$ is an *atomic* value $f(\alpha_1, \ldots, \alpha_k)$, and not a sequence (as defined in part 5). The atomic values $\alpha_1, \ldots, \alpha_k$ are defined as follows. For each $i = 1, \ldots, k$, if $\text{PEVAL}(t_i, x)$ is an atomic value, then α_i is this value, after proper conversion. If $\text{PEVAL}(t_i, x)$ is a sequence, then α_i is the *first* element in this sequence, after proper conversion. For example, if the query is $Q = $ `/a[b + 2 = 5]` and the document is $D = \langle\text{a}\rangle\langle\text{b}\rangle0\langle/\text{b}\rangle\langle\text{b}\rangle3\langle/\text{b}\rangle\langle/\text{a}\rangle$, then according to the standard specification, the predicate evaluates to `false`, because the first "b" child of the "a" node does not satisfy the predicate. Under our definition, however, the predicate will evaluate to `true`, because the second "b" child of the "a" node satisfies the predicate. Our results can be modified to work also for the standard specification of XPath, yet with an extra layer of technical details. In order to keep our presentation more clean, we chose to use the above definition.

The evaluation of a query on a document is defined as follows:

Definition 3.6 (Query evaluation). The *evaluation of a query Q on a document D*, denoted $\text{FULLEVAL}(Q, D)$, is defined to be $\text{SELECT}(\text{OUT}(Q) | \text{ROOT}(Q) = \text{ROOT}(D))$, if $\text{ROOT}(D)$ satisfies the predicate of $\text{ROOT}(Q)$, and the empty sequence otherwise. We say that D *matches* Q, if $\text{FULLEVAL}(Q, D) \neq \emptyset$. We denote by BOOLEVAL the boolean version of FULLEVAL: $\text{BOOLEVAL}(Q, D) = $ `true` if and only if D matches Q. As mentioned in the introduction, for any fixed query Q, FULLEVAL_Q and BOOLEVAL_Q are functions on documents, defined as: $\text{FULLEVAL}_Q(D) = \text{FULLEVAL}(Q, D)$ and $\text{BOOLEVAL}_Q(D) = \text{BOOLEVAL}(Q, D)$.

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()` (also denoted $\langle\$\rangle$).

2. `endDocument()` (also denoted $\langle/\$\rangle$).

3. `startElement(n)`, where $n \in \mathcal{N}$ (also denoted $\langle n \rangle$).

4. `endElement(n)` (also denoted $\langle/n\rangle$).

5. `text(`α`)`, where $\alpha \in \mathcal{S}$ (also denoted α).

If an element is empty, we will use the notation $\langle n/\rangle$ as a shorthand for $\langle n \rangle\langle/n\rangle$.

3.2   Communication complexity

In the communication complexity model [29, 23] 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.

Let $\mathcal{X}$ and $\mathcal{Z}$ be some arbitrary finite sets. Lemma 3.7 below shows that for any function $g : \mathcal{X}^* \to \mathcal{Z} \cup \{\bot\}$, $CC(g')$ 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 3.7 (Reduction lemma). *For any function $g : \mathcal{X}^* \to \mathcal{Z} \cup \{\bot\}$ and for any integer $k \geq 2$, any streaming algorithm computing g requires at least $(CC(g^k) - \log|\mathcal{Z}|)/(k-1)$ bits of memory.*

The proof is rather standard (cf. [23]), 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 $(k-1) \cdot S + \log |\mathcal{Z}|$ bits of communication. It would then immediately follow that $S \geq (CC(g^k) - \log |\mathcal{Z}|)/(k-1)$.

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. Alice and Bob exchange exactly $p + q - 1 = k - 1$ messages of length S and the last message is of length $\log |\mathcal{Z}|$. Thus the total communication of the protocol is $S \cdot (k-1) + \log |\mathcal{Z}|$. $\qquad\square$

Lemma 3.7 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 3.8 (Fooling set). 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 3.9 (Fooling set technique). *Let S be any fooling set for f. Then, $CC(f) \geq \log |S|$.*

The proof appears in Chapter 1 of [23], 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. $\qquad\square$

4   Space lower bounds: simplified version

Our lower bounds are proven with respect to a broad class of XPath queries, the "redundancy-free" queries. Yet, defining Redundancy-free XPath is by itself a major effort, and the definition may seem somewhat contrived at first. In order to give the reader a flavor of the lower bound proofs right away, before we delve into the intricacies of Redundancy-free XPath, we start with a restricted version of the lower bounds. In this section we provide proofs of the three bounds not with respect to arbitrary redundancy-free queries, but rather with respect to three carefully chosen specific queries. The proofs are simpler than the proofs of the general bounds, which appear in Section 7, yet consist of most of the core ideas.

4.1   Query frontier size

We begin with an intuitive overview. Consider any algorithm that evaluates a query Q on a document D, and suppose $x \in D$ is the node whose `startElement` event is currently read from the stream. Let u be a node in Q that x can potentially "match". Whether x will turn into a match of u or not 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 been yet determined 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 4.1 (Frontier size). 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 $\mathrm{F}(x)$, consists of x and of all of its super-siblings. The *frontier size* of T is $\mathrm{FS}(T) = \max_x |\mathrm{F}(x)|$.

Remark. When we discuss frontiers of document trees, we ignore `text` nodes.

Example. Consider the query $Q = $ /a[c[.//e and f] and b > 5] (see Figure 3). The nodes named "b","e", and "f" constitute the frontier at the node named "e". Since this node is the one with the largest frontier, the size of the frontier of this query is 3.
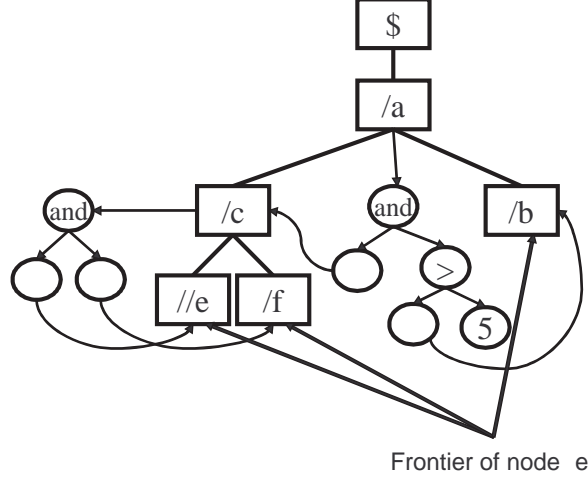

Figure 3: Largest frontier of the query /a[c[.//e and f] and b > 5]

We will prove the following lower bound w.r.t. to evaluation of the above example query Q on XML document streams. In Section 7.1, we generalize the lower bound to arbitrary redundancy-free queries.

Theorem 4.2. *Let* $Q = $*/a[c[.//e and f] and b > 5].* *Then, for every streaming algorithm that computes* BOOLEVAL$_Q$*, there is at least one document on which the algorithm requires at least* $FS(Q) = 3$ *bits of space.*

Proof. We create from the function BOOLEVAL$_Q$ a two-argument function BOOLEVAL$_Q^2$ as described in Section 3.2: the first argument is a prefix of an XML stream and the second argument is a suffix of an XML stream. We will describe a family of documents w.r.t. which there is a $FS(Q)$ lower bound on the communication complexity of BOOLEVAL$_Q^2$. It would follow (Lemma 3.7) that any streaming algorithm evaluating BOOLEVAL$_Q$ needs to use at least $FS(Q)$ bits of space on at least one of the documents in the family.

Let D be the following document (see also Figure 4(a)).

$$D = \langle\texttt{a}\rangle \ \langle\texttt{c}\rangle\langle\texttt{e/}\rangle\langle\texttt{f/}\rangle\langle\texttt{/c}\rangle \ \langle\texttt{b}\rangle\texttt{6}\langle\texttt{/b}\rangle \ \langle\texttt{/a}\rangle.$$

Let x_a, x_b, x_c, x_e, x_f be the nodes named "a","b","c","e", and "f", respectively. Note that this document matches Q. Furthermore, its largest frontier, at x_e, consists of the nodes $\{x_e, x_f, x_b\}$. Thus, $FS(D) = FS(Q)$.

15

(a) The document D (b) The document D_T (c) The document $D_{T,T'}$
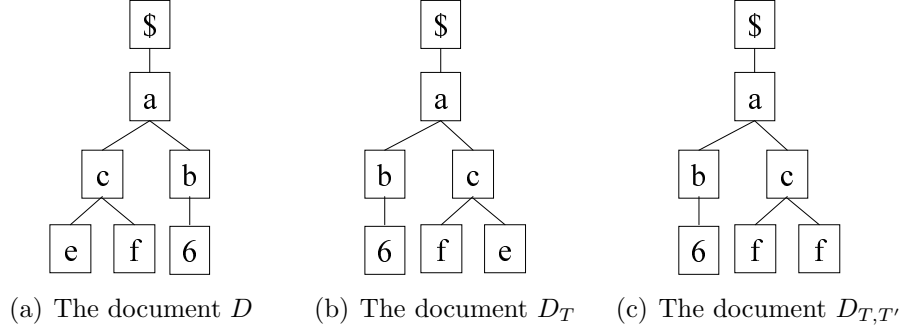
Figure 4: Documents used in the proof of Theorem 4.2

We use the fooling set technique (see Section 3.2) to prove the lower bound for $\textsc{booleval}_Q^2$. We construct a set S of $2^{\mathrm{FS}(D)}$ 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 "similar" to the document D.

We associate with each subset T of $\mathrm{F}(x_e)$ (the frontier at x_e) a pair (α_T, β_T) in S. Thus, $|S| = 2^{|\mathrm{F}(x_e)|} = 2^{\mathrm{FS}(D)}$, as desired.

Recall that $\mathrm{F}(x_e) = \{x_e, x_f, x_b\}$. For each $T \subseteq \mathrm{F}(x_e)$, we define a document D_T, which is the same as D, except that we reorder the children of each node in D, so that in the stream representation of D_T the nodes in T appear before the nodes in $\mathrm{F}(x_e) \setminus T$. For example, if $T = \{x_b, x_f\}$, then D_T is the following document (see also Figure 4(b)).

$$D_T = \langle \mathsf{a} \rangle \ \langle \mathsf{b} \rangle \mathsf{6} \langle / \mathsf{b} \rangle \ \langle \mathsf{c} \rangle \langle \mathsf{f} / \rangle \langle \mathsf{e} / \rangle \langle / \mathsf{c} \rangle \ \langle / \mathsf{a} \rangle.$$

α_T is the prefix of the stream representation of D_T that ends after the $\mathtt{endElement}$ of the last node in T, and β_T is the complementing suffix of the stream. In the above example:

$$\alpha_T = \langle \mathsf{a} \rangle \ \langle \mathsf{b} \rangle \mathsf{6} \langle / \mathsf{b} \rangle \ \langle \mathsf{c} \rangle \langle \mathsf{f} / \rangle \qquad \beta_T = \langle \mathsf{e} / \rangle \langle / \mathsf{c} \rangle \ \langle / \mathsf{a} \rangle.$$

For every two subsets $T, T' \subseteq \mathrm{F}(x_e)$, we let $D_{T,T'}$ be the document whose stream representation is $\alpha_T \circ \beta_{T'}$. It is easy to check that $D_{T,T'}$ is well-formed, since the proper nesting of elements is maintained. For example, if $T = \{x_b, x_f\}$ and $T' = \{x_b, x_e\}$, then $D_{T,T'}$ is the following document (see also Figure 4(c)):

$$D_{T,T'} = \langle \mathsf{a} \rangle \ \langle \mathsf{b} \rangle \mathsf{6} \langle / \mathsf{b} \rangle \ \langle \mathsf{c} \rangle \langle \mathsf{f} / \rangle \ \langle \mathsf{f} / \rangle \langle / \mathsf{c} \rangle \ \langle / \mathsf{a} \rangle.$$

The two following claims establish that S is indeed a fooling set, completing the proof of the theorem.

Claim 4.3. *For every T, D_T matches Q.*

Proof. The query Q is indifferent to how children of nodes in a document are ordered. Hence, since D matches Q, also D_T matches Q. □

Claim 4.4. *For every $T \neq T'$, at least one of $D_{T,T'}, D_{T',T}$ does not match Q.*

Proof. 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 $T \cup (\mathrm{F}(x_e) \setminus T')$ is a proper subset of $\mathrm{F}(x_e)$. Hence, there is a node $z \in \mathrm{F}(x_e)$, which does not belong to $T \cup (\mathrm{F}(x_e) \setminus T')$.

Note that $D_{T,T'}$ includes a node whose name is "b" if and only if $x_b \in T \cup (\mathrm{F}(x_e) \setminus T')$. Similarly, it includes nodes named "e","f" if and only if x_e, x_f, respectively, belong to $T \cup (\mathrm{F}(x_e) \setminus T')$. Now, since there is a node $z \in \{x_b, x_e, x_f\}$, which does not belong to $T \cup (\mathrm{F}(x_e) \setminus T')$, then at least one of the names "b","e","f" is absent from $D_{T,T'}$. Any document that matches Q must have at least one node named "b", one node named "e", and one node named "f". We conclude that $D_{T,T'}$ cannot match 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 3.9) to the function $\textsc{booleval}_Q^2$ and by the reduction lemma (Lemma 3.7). □

Extending the above proof to arbitrary queries is not possible. For example, if we slightly modify Q as follows: $Q' = $ `/a[c[.//* and f] and b > 5]`, then the query frontier size is no longer a correct lower bound. $\mathrm{FS}(Q')$ is still 3, but since any node that matches the nodes named "f" also matches the wildcard node, then only 2 bits of space are sufficient to evaluate the query. Redundancy-free queries do not allow the same document node to match multiple query nodes simultaneously, and thus we can prove that the query frontier size holds for them.

4.2   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 match v. For example, if Q is `//a[b and c]` and D is $\langle a \rangle \langle a \rangle \langle b/ \rangle \langle c/ \rangle \langle /a \rangle \langle /a \rangle$, then the recursion depth of D w.r.t. the node named `a` is 2.

In this section we prove that for the query $Q = $`//a[b and c]`, the document recursion depth is a lower bound on the space complexity of $\textsc{booleval}_Q$ in the data stream model. In Section 7.2 we extend this proof to arbitrary redundancy-free queries that contain queries like Q as a "sub-query".

Theorem 4.5. *Let $Q = //a[b$ and $c]$, and let v be the node named "a" in Q. Then, for any streaming algorithm that computes* BOOLEVAL$_Q$*, and for any integer $r \geq 1$, there is at least one document of recursion depth at most r w.r.t. v, on which the algorithm requires $\Omega(r)$ bits of space.*

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 $\mathbf{t}_i = 1$ if and only if $i \in T$). DISJ$(\mathbf{s}, \mathbf{t}) = 1$ if and only if $S \cap T \neq \emptyset$. The communication complexity of DISJ is $\Omega(r)$ (cf. [23]).

We will prove that given a streaming algorithm that computes BOOLEVAL$_Q$, and given any integer $r \geq 1$, if the algorithm uses at most C bits of space on any document of recursion depth at most r w.r.t. v, then we can design a communication protocol that solves the set disjointness problem with C bits of communication. It would then immediately follow that C has to be at least $\Omega(r)$.

To this end we associate with each input pair $(\mathbf{s}, \mathbf{t})$ of DISJ a document $D_{\mathbf{s},\mathbf{t}}$ as follows. $D_{\mathbf{s},\mathbf{t}}$ has r nodes named "a" nested within each other. Each of these "a" nodes may have a left "b" child (i.e., a child named "b" that appears before the nested "a" child) and/or a right "c" child (i.e., a child named "c" that appears after the nested "a" child). The i-th "a" node has a left "b" child if and only if $\mathbf{s}_i = 1$ and it has a right "c" child if and only if $\mathbf{t}_i = 1$. For example, if $r = 3$, $\mathbf{s} = 110$, and $\mathbf{t} = 010$, then $D_{\mathbf{s},\mathbf{t}}$ is defined as follows (see also Figure 5):

$$D_{\mathbf{s},\mathbf{t}} = \langle a \rangle \langle b/ \rangle \langle a \rangle \langle b/ \rangle \langle a \rangle \langle /a \rangle \langle c/ \rangle \langle /a \rangle \langle /a \rangle.$$

It is easy to check that $D_{\mathbf{s},\mathbf{t}}$ matches Q if and only if at least one of the "a" nodes in $D_{\mathbf{s},\mathbf{t}}$ has both a "b" child and a "c" child. This in turn happens if and only if there is some $i \in \{1, \ldots, r\}$ s.t. $\mathbf{s}_i = \mathbf{t}_i = 1$. In other words, $D_{\mathbf{s},\mathbf{t}}$ matches Q iff DISJ$(\mathbf{s}, \mathbf{t}) = 1$.

For each document $D_{\mathbf{s},\mathbf{t}}$, let $\alpha_{\mathbf{s},\mathbf{t}}$ be the prefix of the stream representation of $D_{\mathbf{s},\mathbf{t}}$ ending after the `startElement` event of the last nested "a" node. Let $\beta_{\mathbf{s},\mathbf{t}}$ be the complementing suffix of the stream. In the example above:

$$\alpha_{\mathbf{s},\mathbf{t}} = \langle a \rangle \langle b/ \rangle \langle a \rangle \langle b/ \rangle \langle a \rangle \qquad \beta_{\mathbf{s},\mathbf{t}} = \langle /a \rangle \langle c/ \rangle \langle /a \rangle \langle /a \rangle.$$

Note that $\alpha_{\mathbf{s},\mathbf{t}}$ depends only on $\mathbf{s}$, while $\beta_{\mathbf{s},\mathbf{t}}$ depends only on $\mathbf{t}$.

The protocol for set disjointness proceeds as follows. Alice runs the given streaming algorithm (that evaluates Q) on the XML stream prefix $\alpha_{\mathbf{s},\mathbf{t}}$ (which she can construct, since this prefix depends only on her input $\mathbf{s}$). When she is done, she sends the state of the algorithm to Bob. Bob can continue the execution of the algorithm on the suffix $\beta_{\mathbf{s},\mathbf{t}}$ (again,
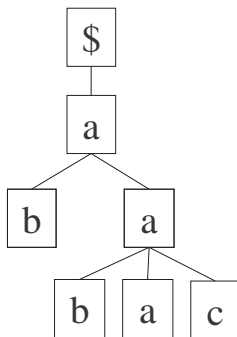
Figure 5: The document $D_{110,010}$

he can construct this suffix, since the suffix depends only on his input $\mathbf{t}$). 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.

Since the document $D_{\mathbf{s},\mathbf{t}}$ is of recursion depth at most r w.r.t. v, the state of the algorithm requires at most C bits to describe, and hence the protocol uses at most C bits of communication. By what we have shown above this protocol correctly computes the function DISJ. Hence, we obtain $C = \Omega(r)$ from the lower bound for DISJ. $\qquad\square$

This relatively simple proof becomes very intricate when we wish to extend it to arbitrary redundancy-free queries that contain queries like Q as a sub-query. The details are provided in Section 7.2.

4.3   Document depth

The *depth* of a document is the length of the longest root-to-leaf path in the tree representing the document. In this section we show that even for evaluating the simple query $Q =$/a/b, any streaming algorithm needs to use space proportional to the logarithm of the document depth (basically meaning that the algorithm has to record the "level" of the elements it scans from the input document). In Section 7.3 we extend the lower bound for evaluation of arbitrary redundancy-free queries that contain queries like Q as a sub-query.

Theorem 4.6. *Let $Q =$/a/b. Then, for any streaming algorithm that evaluates* BOOLEVAL$_Q$, *and for any integer $d \geq 2$, there is at least one document of depth at most d, on which the algorithm requires $\Omega(\log d)$ bits of space.*

Proof. We create from BOOLEVAL$_Q$ a two-argument function BOOLEVAL$_Q^3$ (recall our notations from Section 3.2): its first argument is a pair (α, γ), where α is a prefix of an XML

19

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 3.2. We thus need to create a set S of d documents $D_0, \ldots, D_{d-1}$ of depth at most d that match Q. We then split each document D_i into three parts: $\alpha_i, \beta_i,$ 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.

For each $i = 0, \ldots, d-1$, let D_i be the following document (see also Figure 6(a)):

$$D_i = \langle \mathsf{a} \rangle \underbrace{\langle \mathsf{Z} \rangle \langle \mathsf{Z} \rangle \ldots \langle \mathsf{Z} \rangle}_{i \text{ times}} \underbrace{\langle /\mathsf{Z} \rangle \langle /\mathsf{Z} \rangle \ldots \langle /\mathsf{Z} \rangle}_{i \text{ times}} \langle \mathsf{b} \rangle \langle /\mathsf{b} \rangle \underbrace{\langle \mathsf{Z} \rangle \langle \mathsf{Z} \rangle \ldots \langle \mathsf{Z} \rangle}_{i \text{ times}} \underbrace{\langle /\mathsf{Z} \rangle \langle /\mathsf{Z} \rangle \ldots \langle /\mathsf{Z} \rangle}_{i \text{ times}} \langle /\mathsf{a} \rangle.$$

Note that for all i, the node named "b" in D_i is a child of the node named "a", and thus D_i matches Q. Furthermore, D_i is of depth $\max\{i+1, 2\} \leq d$.

(a) The document D_i (b) The document $D_{i,j}$
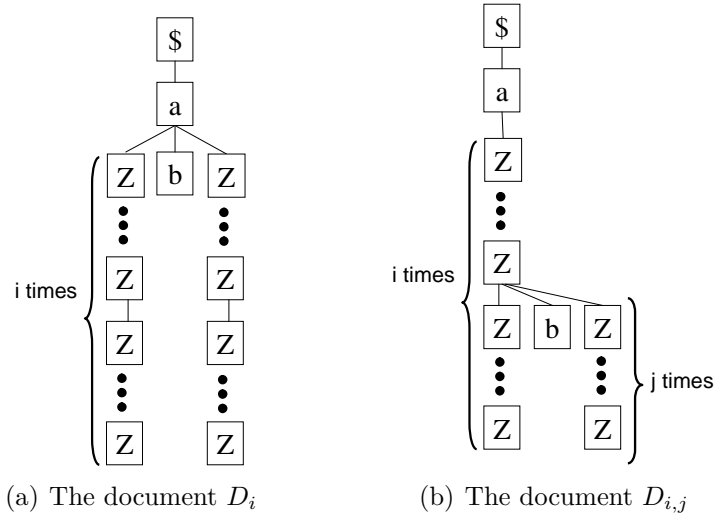
Figure 6: Documents used in the proof of Theorem 4.6

We split the event stream representing D_i into three parts:

1. $\alpha_i = \langle \mathsf{a} \rangle \underbrace{\langle \mathsf{Z} \rangle \langle \mathsf{Z} \rangle \ldots \langle \mathsf{Z} \rangle}_{i \text{ times}}.$

2. $\beta_i = \underbrace{\langle /\mathsf{Z} \rangle \langle /\mathsf{Z} \rangle \ldots \langle /\mathsf{Z} \rangle}_{i \text{ times}} \langle \mathsf{b} \rangle \langle /\mathsf{b} \rangle \underbrace{\langle \mathsf{Z} \rangle \langle \mathsf{Z} \rangle \ldots \langle \mathsf{Z} \rangle}_{i \text{ times}}.$

3. $\gamma_i = \underbrace{\langle /\mathsf{Z} \rangle \langle /\mathsf{Z} \rangle \ldots \langle /\mathsf{Z} \rangle}_{i \text{ times}} \langle /\mathsf{a} \rangle.$

For each $i > j$, let $D_{i,j} \stackrel{\text{def}}{=} \alpha_i \circ \beta_j \circ \gamma_i$. That is $D_{i,j}$ looks as follows (see also Figure 6(b)):

$$D_{i,j} = \langle\texttt{a}\rangle \underbrace{\langle\texttt{Z}\rangle\langle\texttt{Z}\rangle\ldots\langle\texttt{Z}\rangle}_{i \text{ times}} \underbrace{\langle/\texttt{Z}\rangle\langle/\texttt{Z}\rangle\ldots\langle/\texttt{Z}\rangle}_{j \text{ times}} \langle\texttt{b}\rangle\langle/\texttt{b}\rangle \underbrace{\langle\texttt{Z}\rangle\langle\texttt{Z}\rangle\ldots\langle\texttt{Z}\rangle}_{j \text{ times}} \underbrace{\langle/\texttt{Z}\rangle\langle/\texttt{Z}\rangle\ldots\langle/\texttt{Z}\rangle}_{i \text{ times}} \langle/\texttt{a}\rangle.$$

Note that the node named "$\texttt{b}$" becomes the child of the $(i - j)$-th node on the path of $\texttt{Z}$ nodes. The proper nesting of elements is maintained, and therefore $D_{i,j}$ is a well-formed document. Since the "$\texttt{b}$" node is no longer a child of the "$\texttt{a}$" node, $D_{i,j}$ does not match Q.

We conclude that S is indeed a fooling set of size d. Applying Theorem 3.9 to the function BOOLEVAL_Q^3 and Lemma 3.7 give us a space lower bound of $(\log d)/2 = \Omega(\log d)$. $\qquad\square$

As before, extending the above proof to hold for arbitrary redundancy-free queries that contain a query like /a/b as a sub-query is quite intricate. The details are provided in Section 7.3.

5   Redundancy-free XPath

Our general lower bounds deal with generic queries rather than specific queries. This makes arguments much more complicated. Since we don't know much about the query, it is hard to construct generic documents that are guaranteed to match or not to match the query. As shown in the simplified versions of the lower bounds, constructing such documents is an essential part of the proofs.

To address these difficulties, we restrict to queries taken from a fragment of XPath, which we call *Redundancy-free XPath*. The particular properties of queries in Redundancy-free XPath allow us to associate with each query in this fragment a generic "canonical document" (see Section 6.4). A canonical document is guaranteed to match its corresponding query, and furthermore there is a *unique* way to construct this matching. Canonical documents play a crucial role in all our proofs and are used as a basis for constructing the documents that match/don't match the given generic query.

One of the central qualities of queries in Redundancy-free XPath is "minimality": these queries do not consist of redundant parts that can be eliminated without changing the semantics of the queries. For example, the query /a[b > 5 and b > 6]) is not redundancy-free, because the atomic predicate "b > 5" is redundant.

Redundancy-free XPath has other restrictions: queries cannot consist of disjunctions or negations, their atomic predicates can point to only a single variable (e.g., a predicate "[a > b]" is not allowed), they cannot mix wildcard node tests with descendant axis (e.g., expressions like "[a//*]" are disallowed), and they do not allow predicates that restrict values of internal nodes (e.g., predicates like "[a[b] > 5]" are not allowed). All these

restrictions are required for the construction of proper canonical documents, as described in Section 6.4. Yet, we do not try to argue that all of these requirements are necessary for the correctness of the lower bounds. It may be possible that via other techniques one could extend the lower bounds to hold for a larger fragment of XPath.

A possible criticism of Redundancy-free XPath is that some of its restrictions are artificial and non-intuitive. Indeed, Redundancy-free XPath was defined the way it is to allow for the construction of canonical documents. Even so, Redundancy-free XPath is an extensive subclass of XPath, consisting of many natural queries that come up in reality. In fact majority of the queries in the XQuery Use Cases document [9] are composed of Redundancy-free XPath expressions. This is due to the fact that most of the restrictions would not appear very often in human written queries since the users usually write queries as short as possible over known schemas. Examples of restrictions that rarely conflict with queries in the Xquery Use Cases document are redundant predicates, a "*" with a descendant axis, and predicates over internal nodes. Regarding the same set of queries it seems like the most severe restriction of Redundancy-free XPath are the limitations to conjunctive queries and to a lesser extent univariate predicates and this should be the focus of future investigation in this field. Beyond these queries, Redundancy-free XPath might not suffice for tool generated queries. Currently we do not have a large set of such queries available, so we postpone analysis of such cases until XQuery and XPath become more prevalent in query generation tools.

Redundancy-free XPath is defined as follows:

Definition 5.1 (Redundancy-free queries). A Forward XPath query is called *redundancy-free*, if it is: (1) star-restricted; (2) conjunctive; (3) univariate; (4) leaf-only-value-restricted; and (5) strongly subsumption-free.

Remark. In the preliminary version of the paper [4] this fragment of XPath is referred to as "Conjunctive XPath".

In the following we formally define these restrictions and state useful properties they have. The interested reader can find full proofs in Appendix A.

5.1   Star-restricted queries

Definition 5.2 (Star-restricted query). A query Q is called *star-restricted*, if none of the nodes in Q that have a wildcard node test are: (1) leaves; (2) have a descendant axis; or (3) have a child with a descendant axis.

That is, in star-restricted queries path expressions, such as `a/*`, `a//*/b`, and `a/*//b`, are disallowed.

5.2   Conjunctive queries

In order to define conjunctive XPath queries, we need to define "atomic predicates":

Definition 5.3 (Atomic predicate). A predicate subexpression is called an *atomic predicate*, if it satisfies the two following conditions:

1. None of its nodes is labeled by a function or operator on boolean arguments (such as the logical operators `and,or,not`).

2. None of its nodes, except for the root, is labeled by a function or operator whose output is boolean.

Example. In the predicate `[b > 5 and c + d = 7]`, the subexpressions "`b > 5`" and "`c + d = 7`" are atomic predicates.

Definition 5.4 (Conjunctive query). A predicate is called *conjunctive* if it is either an atomic predicate or a conjunction of atomic predicates. A query is called *conjunctive*, if all its predicates are conjunctive.

That is, the only function on boolean arguments allowed in conjunctive queries is the logical `and`. Furthermore, expressions in which nodes labeled by functions with boolean output are children of nodes labeled by functions on non-boolean arguments (thereby necessitating casting of boolean to non-boolean) are disallowed. An example of such an expression is `1 - (a > 5)`.

5.3   Univariate queries

Definition 5.5 (Univariate query). A *variable* in an atomic predicate is a reference to a query node. An atomic predicate is called *univariate*, if it consists of at most one variable. A conjunctive predicate is called *univariate*, if all its constituent atomic predicates are univariate. A conjunctive query is called *univariate*, if all its predicates are univariate.

Example. In the conjunctive predicate `[b > 5 and c + d = 7]`, the first atomic predicate "`b > 5`" is univariate, while the second "`c + d = 7`" is not.

Note that by the definition of predicates, successor nodes can never be pointed by predicates. It follows that a predicate of the form `[a//b]` is univariate, although it refers to two query nodes. Only the "`a`" node is a real variable, because the "`b`" node is a successor.

A special property of univariate queries is that their nodes can be associated with "truth sets", as defined below.

Definition 5.6 (Truth set). Let P be a univariate atomic predicate. Since P has a variable, its value (`true` or `false`) is undetermined. A value $\alpha \in \mathcal{V}$ is said to *satisfy* P, if replacement of the variable of P by α results in a tautology (i.e., the predicate evaluates to `true`). The *truth set* of P, denoted $\text{TRUTH}(P)$, is the set of all string values in $\mathcal{S}$, which, after proper casting to the required type, satisfy P.

For a node u in a univariate query Q, let v be the succession root of u. The *truth set* of u, denoted $\text{TRUTH}(u)$, is defined as follows:

1. If u is a succession leaf and v is a variable in an atomic predicate P, then $\text{TRUTH}(u) = \text{TRUTH}(P)$.

2. If u is a succession leaf and v is not a variable in any predicate (i.e., $v = \text{ROOT}(Q)$), then $\text{TRUTH}(u) = \mathcal{S}$.

3. If u is not a succession leaf, then $\text{TRUTH}(u) = \mathcal{S}$.

Example. In the query `/a[b/c > 5 and d]`, the truth set of the nodes named "a", "b" and "d" is $\mathcal{S}$, while the truth set of the node named "c" is the set of strings representing numbers in the interval $(5, \infty)$.

5.4 Leaf-only-value-restricted queries

Definition 5.7 (Leaf-only-value-restricted queries). Let Q be a univariate query. A node $u \in Q$ is called *value-restricted*, if $\text{TRUTH}(u)$ is a proper subset of $\mathcal{S}$ (i.e., $\text{TRUTH}(u) \subsetneq \mathcal{S}$). Q is called *leaf-only-value-restricted*, if none of its internal nodes is value-restricted.

Example. The query `/a[b[c] > 5]` is not leaf-only-value-restricted, because the node named "b" is internal but value-restricted. On the other hand, the query `/a[b[c > 5]]` is leaf-only-value-restricted.

5.5 Subsumption-free queries

Loosely speaking, subsumption-free queries are ones that do not have any "redundancies" (i.e., removal of any part of a subsumption-free query results in a query which is not equivalent to the original query). For our proofs, we will need a rather strong notion of subsumption-freeness, which we gradually develop below. To this end, we define subsumption-freeness only w.r.t. queries that are: (1) star-restricted; (2) conjunctive; (3) univariate; and (4) leaf-only-value-restricted.

Our most powerful tool for proving whether a document matches a query or not is the notion of *matchings* defined below. Matchings will be also useful for formally defining subsumption-free queries.

Definition 5.8 (Matching). Let Q be a univariate query, and let D be any document. A *matching of a node $x \in D$ with a node $u \in Q$* is a mapping ϕ from the node set of Q_u to the node set of D_x (recall our conventional notation), which has the following properties:

1. **Root match:** $\phi(u) = x$.

2. **Axis match:** For all nodes $v \in Q_u$, $v \neq u$, $\phi(v)$ relates to $\phi(\text{PARENT}(v))$ according to $\text{AXIS}(v)$.

3. **Node test match:** For all nodes $v \in Q_u$, $\text{NAME}(\phi(v))$ passes $\text{NTEST}(v)$.

4. **Value match:** For all nodes $v \in Q_u$, $\text{STRVAL}(\phi(v)) \in \text{TRUTH}(v)$.

If a mapping ϕ satisfies only the first three requirements, we call it a *structural matching*.

A *matching* (resp., *structural matching*) of the document D and the query Q is a matching (resp., structural matching) of $\text{ROOT}(D)$ with $\text{ROOT}(Q)$.

Example. Figure 7 shows two example matchings of a document with a query. In this case the "b" node in the query can be matched with any of the two "b" nodes in the document whose string value belongs to its truth set.
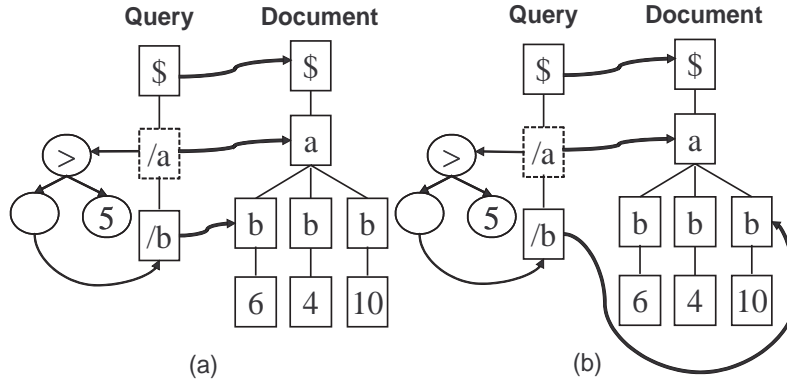
Figure 7: Two matchings of a document with the query `/a[b > 5]`

The following definition and lemma show that matchings characterize the set of nodes selected by a query node.

Definition 5.9 (Matching relative to a context). Let Q be a univariate query and let D be a document. Let $u \in Q, x \in D$ be any nodes, and let $v \in Q_u, y \in D_x$. y is said to *match (resp., structurally match) v relative to the context $u = x$*, if there is a matching (resp., structural matching) ϕ of x with u, so that $\phi(v) = y$.

25

Remark. For the rest of the paper, when we say that a document node x matches a query node u, without specifying a context, we refer to the context $\text{ROOT}(Q) = \text{ROOT}(D)$.

Lemma 5.10. *A document D matches a query Q if and only if there exists a matching of D and Q.*

Definition 5.11 (Set of matches). The *set of matches (resp., structural matches) for a node $u \in Q$*, denoted $\text{MATCHES}(u)$ (resp., $\text{SMATCHES}(u)$), is the set of all pairs $\langle D, x \rangle$, where D is a document, $x \in D$ is a node in this document, and x matches (resp., structurally matches) u relative to the context $\text{ROOT}(Q) = \text{ROOT}(D)$.

Definition 5.12 (Subsumption). A node $u \in Q$ is said to *subsume* (resp., *structurally subsume*) a node $v \in Q$, if $\text{MATCHES}(u) \subseteq \text{MATCHES}(v)$ (resp., $\text{SMATCHES}(u) \subseteq \text{SMATCHES}(v)$).

Example. In the query `/a[b and .//b]`, the left node named "`b`" subsumes the right one, because any document node that would match the left "`b`" node will also match the right one. On the other hand, in the query `/a[b = 5 and .//b = 3]`, the left node named "`b`" structurally subsumes the right one, but does not subsume it.

The following is an extension of the notion of subsumption:

Definition 5.13 (Subsumption of sets). A node $u \in Q$ is said to *subsume a set of nodes $v_1, \ldots, v_k \in Q$*, if $\text{MATCHES}(u) \subseteq \bigcup_{i=1}^{k} \text{MATCHES}(v_i)$.

Example. In the query `/a[fn:matches(b,''^A.*B$'') and fn:matches(b,''AB'') and fn:matches(b,''A.+B'')]`, the three nodes named "`b`" structurally subsume each other. The truth set of the first node consists of all the strings that start with "`A`" and end with "`B`"; the truth set of the second node consists of all the strings that contain "`AB`" as a substring; the truth set of the third node consists of all the strings that contains substrings of length at least 3 that start with "`A`" and end with "`B`". It follows that none of the three nodes individually subsumes each other, but the first node subsumes the set consisting of the second and the third nodes.

We can now define subsumption-free queries:

Definition 5.14 (Subsumption-free query). Q is *subsumption-free*, if no node $u \in Q$ subsumes any set $S \subseteq Q \setminus \{u\}$.

In order to define the stronger notion of subsumption-freeness we need for our proofs, we present two properties of queries. To this end, we will use the following notion:

Definition 5.15 (Domination set). The *domination set* (resp., *structural domination set*) of a node $u \in Q$, denoted DOM(u) (resp., SDOM(u)) is the set of all nodes $v \in Q$ that u subsumes (resp., structurally subsumes).

In the two following definitions, for each node u, we denote by L_u the set of **leaf** nodes in its structural domination set.

Definition 5.16 (Sunflower property). Q is said to have the *sunflower property*, if every **leaf** node $u \in Q$ satisfies the following:

$$\text{TRUTH}(u) \not\subseteq \bigcup_{v \in L_u} \text{TRUTH}(v).$$

For a string α, we denote by PREFIX(α) the set of all prefixes of α. For a set of strings T, we denote by PREFIX(T) the set of all prefixes of all strings $\alpha \in T$.

Definition 5.17 (Prefix sunflower property). Q is said to have the *prefix sunflower property*, if every **internal** node $u \in Q$ satisfies the following:

$$\text{PREFIX}(\text{TRUTH}(u)) \not\subseteq \bigcup_{v \in L_u} \text{PREFIX}(\text{TRUTH}(v)).$$

We can now define strongly subsumption-free queries:

Definition 5.18 (Strongly subsumption-free queries). Let Q be a star-restricted, leaf-only-value-restricted, univariate, conjunctive query. Q is called *strongly subsumption-free* if it has the sunflower property and the prefix sunflower property.

The following shows that strongly subsumption-free queries are indeed subsumption-free:

Lemma 5.19. *If Q is strongly subsumption-free, then it is also subsumption-free.*

The following example shows that strong subsumption-freeness is a strictly stronger notion than subsumption-freeness.

Example. Consider the query `/a[b[c = ''A''] and fn:ends-with(b,''B'')]`. The first "b" node does not subsume the second "b" node, because not every document node that matches the first "b" node must have a string value that ends with the character "B". The second "b" node does not subsume the first "b" node, because not every document node that matches the second "b" node must have a child named "c". Therefore, this query is subsumption-free. However, the query is not strongly subsumption-free: the only node to structurally subsume other nodes is the first "b" node that structurally subsumes the second "b" node. The first "b" node is internal. However, since the truth set of the second "b" node consists of all the strings that end with the character "B", then all strings in $\mathcal{S}$ are prefixes of some string in this truth set. Thus, the query does not have the prefix sunflower property, and therefore is also not strongly subsumption-free.

6 Technical machinery

Our lower bound proofs are based on relatively simple reductions from the model of communication complexity. Yet, proving that the document instances that come out of these reductions match/don't match the given query requires somewhat involved and lengthy arguments.

In order to minimize the amount of details in the lower bound proofs themselves, we collected in this section most of the technical machinery used to argue about matching of documents to queries. We include in this section only non-trivial proofs. The more straightforward (yet laborious) proofs are deferred to the appendix.

6.1 Document homomorphisms

Document homomorphisms are analogous to reductions in algorithms and complexity. Suppose we have a document D that we already know to match a query Q and let D' be another document we wish to prove matches Q. If we show a *homomorphism* from D to D', then we can immediately deduce that D' also matches Q.

Loosely speaking, a homomorphism from D to D' is a mapping from the nodes of D to the nodes of D' that preserves parent-child relationships, node names, and string values. Formally, homomorphisms are defined w.r.t. subtrees of documents:

Definition 6.1 (Document homomorphism). Let D, D' be two documents (possibly, $D = D'$), and let $x \in D$ and $x' \in D'$ be two nodes in these documents. The subtree D_x is said to be *homomorphic* to the subtree $D'_{x'}$, if there is a mapping ξ (called a *homomorphism*) from the node set of D_x to the node set of $D'_{x'}$ that satisfies the following:

1. **Root preservation:** $\xi(x) = x'$.

2. **Tree-relationship preservation:** For each node $y \in D_x$, $y \neq x$, $\xi(\text{PARENT}(y)) = \text{PARENT}(\xi(y))$.

3. **Name preservation:** For each node $y \in D_x$, $\text{NAME}(\xi(y)) = \text{NAME}(y)$.

4. **Value preservation:** For every node $y \in D_x$, $\text{STRVAL}(\xi(y)) = \text{STRVAL}(y)$.

If ξ satisfies only the first three of the above properties, then we call it a *structural homomorphism* and we say that D_x is *structurally homomorphic* to $D'_{x'}$.

If ξ satisfies the value preservation property only for **leaf** nodes y, we call it a *weak homomorphism* and we say that D_x is *weakly homomorphic* to $D'_{x'}$.

A document D is said to be *homomorphic* (resp., *structurally homomorphic, weakly homomorphic*) to the document D', if $D_{\text{ROOT}(D)}$ is homomorphic (resp., structurally homomorphic, weakly homomorphic) to $D'_{\text{ROOT}(D')}$.

Example. Let

$$D' = \langle a \rangle \ \langle b \rangle \ \texttt{hello} \ \langle /b \rangle \ \langle c \rangle \ \texttt{world} \ \langle /c \rangle \ \langle /a \rangle$$

and

$$D = \langle a \rangle \ \langle c \rangle \ \texttt{world} \ \langle /c \rangle \ \langle c \rangle \ \texttt{world} \ \langle /c \rangle \ \langle b \rangle \ \texttt{hello} \ \langle /b \rangle \ \langle /a \rangle.$$

A weak homomorphism from D to D' is one that maps the node named "a" in D to the node named "a" in D', the node named "b" in D to the node named "b" in D', and the two nodes named "c" in D to the node named "c" in D'. This is not a homomorphism, because the string value of the "a" node is not preserved.

The following lemma shows that if D matches Q and is homomorphic to D', then also D' matches Q.

Lemma 6.2. *Let D, D' be two documents, let $x \in D$ and $x' \in D'$ be two nodes in these documents, and assume there is a homomorphism (resp., structural homomorphism) ξ from D_x to $D'_{x'}$. Let Q be a redundancy-free query, and suppose there is a matching (resp., structural matching) ϕ of x with a node $u \in Q$. Then, the mapping $\eta \stackrel{\text{def}}{=} \xi \circ \phi$ is a matching (resp., structural matching) of x' with u.*

The same lemma holds for weak homomorphisms, if we restrict the matching ϕ to the following class of matchings:

Definition 6.3 (Leaf-preserving matchings). Let Q be a univariate query, and let D be any document. A matching ϕ of a node $x \in D$ with a node $u \in Q$ is called *leaf-preserving*, if for every leaf $v \in Q_u$, $\phi(v)$ is a leaf.

Now, we have:

Lemma 6.4. *Let D, D' be two documents, let $x \in D$ and $x' \in D'$ be two nodes in these documents, and assume there is a weak homomorphism ξ from D_x to $D'_{x'}$. Let Q be a redundancy-free query, and suppose there is a leaf-preserving matching ϕ of x with a node $u \in Q$. Then, the mapping $\eta \stackrel{\text{def}}{=} \xi \circ \phi$ is a matching of x' with u.*

The proofs of both lemmas appear in Appendix C.

Definition 6.5 (Isomorphism). A homomorphism ξ from D to D' is called an *isomorphism*, if it is injective and onto.

Remark. It is immediate that if ξ is an isomorphism, then so is ξ^{-1}.

6.2 Hybrid matchings

Hybrid matchings enable the pasting together of two "partial" matchings into a "full" matching. Let Q be a univariate query and let D be some document. Let $u \in Q$ be a node, which is not the root, and suppose there is a matching ϕ of a node $x \in D$ with u.

Let Q_{-u} denote the query Q after removing the subtree rooted at u. Formally speaking, Q_{-u} need not be a legal query by itself, because the predicate of PARENT(u) may point to u. Yet, we can still talk about matchings of documents with Q_{-u}. A mapping η from the nodes of Q_{-u} to the nodes of a document D is called a matching (relative to the context ROOT(Q) = ROOT(D)), if it satisfies the four properties of a matching.

We can now define hybrid mappings:

Definition 6.6 (Hybrid mappings). Let Q, D, u, x be as above. Suppose ϕ is a matching of x with u and η is a matching of D with Q_{-u}. The *hybrid mapping* induced by ϕ and η is a mapping μ from Q to D defined as follows for every $v \in Q$:

$$\mu(v) = \begin{cases} \phi(v) & \text{if } v \in Q_u \\ \eta(v) & \text{if } v \in Q_{-u} \end{cases}.$$

The following lemma, whose proof appears in Appendix B, gives a sufficient condition for the hybrid mapping to indeed be a matching:

Lemma 6.7. *Let Q be a univariate query, let D be a document, let ϕ be a matching of a node $x \in D$ with a node $u \in Q$, and let η be a matching of D with Q_{-u}. If x relates to $\eta(\text{PARENT}(u))$ according to AXIS(u), then the hybrid mapping μ induced by ϕ and η is a matching of D with Q.*

6.3 Query automorphisms

Structural query automorphisms are a tool for characterizing which nodes of a query structurally subsume other nodes.

Definition 6.8 (Structural query automorphism). A mapping ψ from the node set of Q to itself is called a *structural query automorphism*, if it has the following properties:

1. **Root preservation:** $\psi(\text{ROOT}(Q)) = \text{ROOT}(Q)$.

2. **Axis preservation:** For all $u \in Q$, $u \neq \text{ROOT}(Q)$, if AXIS$(u) = \texttt{child}$ (resp., AXIS$(u) = \texttt{descendant}$), then $\psi(u)$ is a child (resp., descendant) of $\psi(\text{PARENT}(u))$, and AXIS$(\psi(u)) = \texttt{child}$ (resp., AXIS$(\psi(u)) \in \{\texttt{child}, \texttt{descendant}\}$).

3. **Node test preservation:** For all $u \in Q$, if NTEST$(u) \neq$ *, then NTEST$(\psi(u)) =$ NTEST(u).

Such an automorphism is called *non-trivial*, if it is not the identity.

Example. In the query `/a[b and .//b]`, a non-trivial structural query automorphism is one that maps the node named "a" to itself, and the two nodes named "b" to the left node named "b".

The following lemma shows that structural query automorphisms characterize the nodes that structurally subsume other nodes. The proof appears in Appendix D.

Lemma 6.9. *A node $u \in Q$ structurally subsumes a node $v \in Q$ if and only if there exists a structural query automorphism ψ on Q, such that $\psi(v) = u$.*

Let DEPTH$(u) = |$PATH$(u)|$ be the number of nodes along the path from the root to u. The following is a property of structural query automorphisms: (The proof is in Appendix D.)

Proposition 6.10. *Let ψ be any structural query automorphism on Q. Then, for all $u \in Q$, DEPTH$(u) \leq$ DEPTH$(\psi(u))$.*

6.4 Canonical documents

In this section we introduce the notion of *canonical documents*. Canonical documents will be one of our primary tools for proving the memory lower bounds. For every redundancy-free query Q, we define a corresponding "canonical document" D_c. This document has certain properties, which will become very handy in our proofs.

The construction Loosely speaking, the canonical document corresponding to a query is identical to the query, except for the following differences: (1) node tests are turned into node names; (2) nodes with `descendant` axis, are made strict descendants of their parents, by inserting a long chain of "artificial nodes" between them and their parents; (3) nodes are assigned string values, which "uniquely" belong to their truth sets.

The function `createCanonicalDocument(Q)` (see Figure 8) describes how to construct a canonical document D_c from a query Q. `getAuxiliaryName(Q)` is a function that returns a name from $\mathcal{N}$, which does not occur as a node test in Q. We assume that $\mathcal{N}$ is large enough so such a name always exists.

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.

We create two types of nodes in D_c: "shadow nodes" and "artificial nodes". For every node $u \in Q$, we create a single shadow node $\textsc{shadow}(u)$ in D_c inductively as follows. First, $\textsc{shadow}(\textsc{root}(Q)) = \textsc{root}(D_c)$. Assume, then, that we defined $\textsc{shadow}(u)$, and let v be a child of u. If $\textsc{axis}(v) = \texttt{child}$, then $\textsc{shadow}(v)$ is set to be a child of $\textsc{shadow}(u)$. If $\textsc{axis}(v) = \texttt{descendant}$, then $\textsc{shadow}(v)$ is set to be a descendant of $\textsc{shadow}(u)$, following a chain of $h+1$ new artificial nodes $z_1, \ldots, z_{h+1}$ (lines 2-6,17-21). The names of $z_1, \ldots, z_{h+1}$ are assigned the auxiliary name returned by $\texttt{getAuxiliaryName}(Q)$. If $\textsc{ntest}(v) \neq *$, then the name of $\textsc{shadow}(v)$ is set to be $\textsc{ntest}(v)$. Otherwise, it is assigned a name returned by $\texttt{getAuxiliaryName}(Q)$ (lines 7-9,16).

The document constructed thus far (i.e., the one created by $\texttt{createCanonicalDocument}$, excluding line 10) has no $\texttt{text}$ nodes. We call such a document a "structurally canonical document".

We next show how to add $\texttt{text}$ nodes to D_c. Only shadow nodes are assigned $\texttt{text}$ node children. Let u be any node in Q. Let L_u be the set of leaf nodes in the structural domination set of u (recall definition from Section 5.5). If u is a leaf, then by the sunflower property of Q, there exists a value $\alpha \in \textsc{truth}(u)$, which does not belong to $\textsc{truth}(v)$, for all $v \in L_u$. We add a text node child to $\textsc{shadow}(u)$, whose text content is α. If u is an internal node, then by the prefix sunflower property of Q, there exists a value $\alpha \in \mathcal{S}$, which is not a prefix of any value in $\bigcup_{v \in L_u} \textsc{truth}(v)$. We add a text node child to $\textsc{shadow}(u)$, preceding all its other children, whose text content is α. The function $\texttt{getUniqueValue(u)}$ (line 10) is the one that returns the "unique value" α, as specified above.

Example. Consider the following redundancy-free query:

$$Q = \texttt{/a[*/b > 5 and c/b//d > 12 and .//d < 30]}.$$

Note that the second "b" node in this query structurally subsumes the first "b" node (which is a leaf) and the first "d" node structurally subsumes the second "d" node (which is also a leaf). The maximum length of a wildcard chain in this query is 1 and "Z" is an "auxiliary name". Therefore, the following is a canonical document corresponding to Q:

$\langle\texttt{a}\rangle\ \langle\texttt{Z}\rangle\langle\texttt{b}\rangle\texttt{6}\langle\texttt{/b}\rangle\langle\texttt{/Z}\rangle\ \langle\texttt{c}\rangle\langle\texttt{b}\rangle\texttt{hello}\langle\texttt{Z}\rangle\langle\texttt{Z}\rangle\langle\texttt{d}\rangle\texttt{31}\langle\texttt{/d}\rangle\langle\texttt{/Z}\rangle\langle\texttt{/Z}\rangle\langle\texttt{/b}\rangle\langle\texttt{/c}\rangle\ \langle\texttt{Z}\rangle\langle\texttt{Z}\rangle\langle\texttt{d}\rangle\texttt{29}\langle\texttt{/d}\rangle\langle\texttt{/Z}\rangle\langle\texttt{/Z}\rangle\ \langle\texttt{/a}\rangle.$

Figure 9 shows the tree representation of the query and the canonical document. The first node named "Z" in the canonical document is the shadow of the wildcard node in Q. The rest of the "Z" nodes are artificial nodes. The shadow of the first "b" node in Q was assigned the value "6", which belongs to the truth set of this node. The shadow of the second "b" node in Q was assigned the string "hello" as a prefix, because no value in the truth set of the first "b" node has "hello" as a prefix. The shadow of the first "d" node

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

Function processNode(u)
1: if (u != root(Q)) then

2:    if (axis(u) = descendant) then
3:      for i := 1 to h + 1 do
4:        print '⟨' getAuxiliaryName(Q) '⟩'
5:      end for
6:    end if

7:    a := ntest(u)
8:    if (a = '*') a := getAuxiliaryName(Q)
9:    print '⟨' a '⟩'

10:   print getUniqueValue(u)

11: end if

12: for c in children(u) do
13:   processNode(c)
14: end for

15: if (u != root(Q)) then

16:   print '⟨/' a '⟩'

17:   if (axis(u) = descendant) then
18:     for i := 1 to h + 1 do
19:       print '⟨/' getAuxiliaryName(Q) '⟩'
20:     end for
21:   end if

22: end if
```

Figure 8: Pseudo-code of the procedure that creates a canonical document for a given query.

was assigned the value "31", because it belongs to the truth set of this "d" node, but does not belong to the truth set of the second "d" node. Finally, the shadow of the second "d" node was assigned the value "29", which belongs to the truth set of this node.

Canonical matching We present a "canonical matching" of D_c and Q, and prove it is unique. To this end, fix Q to be an arbitrary redundancy-free query, and let D_c be the corresponding canonical document. The canonical matching ϕ_c is defined as follows:

$$\text{For every node } u \in Q, \quad \phi_c(u) \stackrel{\text{def}}{=} \text{SHADOW}(u).$$

Lemma 6.11. ϕ_c is a matching of D_c and Q.

Proof. We need to prove ϕ_c satisfies the four properties of a matching (see Definition 5.8).

1. **Root match:** By definition, $\text{SHADOW}(\text{ROOT}(Q)) = \text{ROOT}(D_c)$.
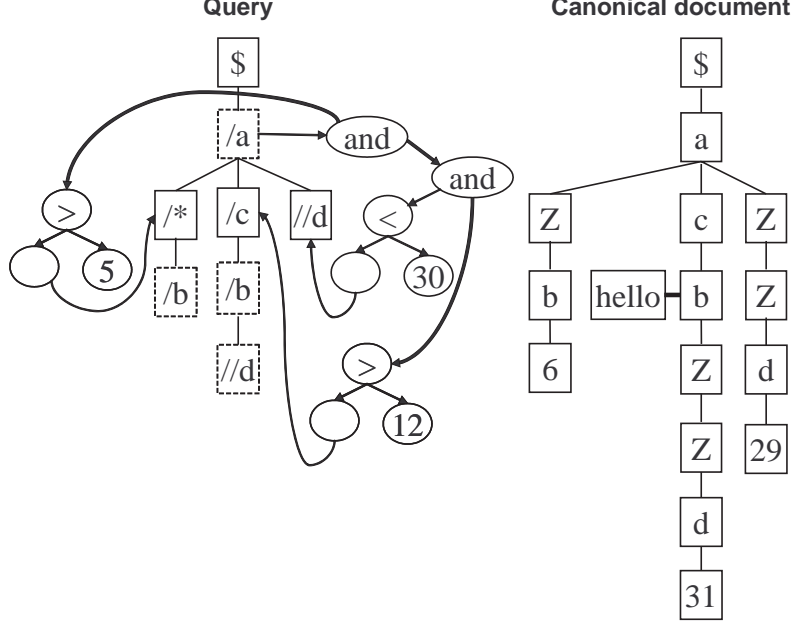
33

Figure 9: Canonical document for `/a[*/b > 5 and c/b//d > 12 and .//d < 30]`

2. **Axis match:** Let u be any node in Q which is not the root. If $\text{AXIS}(u) = \text{child}$, then by the construction of D_c, $\text{SHADOW}(u)$ is a child of $\text{SHADOW}(\text{PARENT}(u))$. If $\text{AXIS}(u) = \text{descendant}$, then by the construction of D_c, $\text{SHADOW}(u)$ is a descendant of $\text{SHADOW}(\text{PARENT}(u))$.

3. **Node test match:** For any node $u \in Q$, if $\text{NTEST}(u) \neq *$, then the name of $\text{SHADOW}(u)$ is the same as $\text{NTEST}(u)$.

4. **Value match:** Let u be any node in Q. If u is a leaf, then $\text{SHADOW}(u)$ has a single text node child whose text content belongs to $\text{TRUTH}(u)$. Therefore, $\text{STRVAL}(\text{SHADOW}(u)) \in \text{TRUTH}(u)$. If u is an internal node, then since Q is leaf-only-value-restricted, $\text{TRUTH}(u) = \mathcal{S}$. That is, every string, $\text{STRVAL}(\text{SHADOW}(u))$ in particular, belongs to $\text{TRUTH}(u)$. $\square$

We next show that in any structural matching of D_c with Q, no node of Q can be mapped to an artificial node in D_c:

Lemma 6.12. *Let ϕ be any structural matching of D_c and Q. Then, for every node $u \in Q$, $\phi(u)$ is not an artificial node.*

We first prove the following claim:

Claim 6.13. *Let $u \in Q$ be any node so that $\phi(u)$ is an artificial node. Then, $\text{NTEST}(u) = *$.*

34

Proof. Since $\phi(u)$ is artificial, its name is the auxiliary name returned by the function `getAuxiliaryName(Q)`. Hence, this name does not occur as a node test of any node in Q, and in particular it cannot be the node test of u. Since the name of $\phi(u)$ passes $\text{NTEST}(u)$, it must be the case that $\text{NTEST}(u) = \texttt{*}$. $\qquad\square$

Proof of Lemma 6.12. Suppose, to the contradiction, there exists some node $u \in Q$ so that $\phi(u)$ is an artificial node. By the above claim, u has a wildcard node test. Since Q is star-restricted (recall definition in Section 5.1), this implies that u must have a `child` axis and must have a child v with a `child` axis.

By our construction of the document D_c, since $\phi(u)$ is an artificial node, it belongs to a chain of $h + 1$ artificial nodes, where h is the length of the longest chain of wildcard nodes in Q. Let $i \in \{1, \ldots, h + 1\}$ be the position of $\phi(u)$ in this chain.

Since u has a `child` axis, then $\phi(u)$ must be a child of $\phi(\text{PARENT}(u))$. If $i > 1$, this means that also $\phi(\text{PARENT}(u))$ is an artificial node, and thus also $\text{NTEST}(\text{PARENT}(u)) = \texttt{*}$. Inductively, this argument implies that the first $i - 1$ ancestors of u (starting from its parent and upwards) must have a wildcard node test.

Since u has a child v with a `child` axis, $\phi(v)$ must be a child of $\phi(u)$. If $i < h + 1$, then $\phi(v)$ is also an artificial node, and thus $\text{NTEST}(v) = \texttt{*}$. Inductively, this argument implies that there must be a sequence of $h + 1 - i$ nodes in Q, starting with v, each is a child of the previous one, and all have a wildcard node test.

We conclude from the above two paragraphs, that u must belong to a chain of $h + 1$ nodes, all of which have a wildcard node test. This contradicts the fact h is the maximum length of a chain of nodes with a wildcard node test in Q. $\qquad\square$

The next lemma shows that any structural matching of D_c with Q induces a corresponding structural query automorphism:

Lemma 6.14. *Let ϕ be any structural matching of D_c and Q. Then, $\psi(u) \stackrel{\text{def}}{=} \text{SHADOW}^{-1}(\phi(u))$ is a structural query automorphism on Q.*

Note that by Lemma 6.12, for every $u \in Q$, $\phi(u) = \text{SHADOW}(v)$, for some $v \in Q$. Note also that $\text{SHADOW}(\cdot)$ is a 1-1 mapping. Therefore, the mapping ψ is well-defined.

Proof. We show that ψ has the three required properties:

1. Root preservation: By the root match property of ϕ, $\phi(\text{ROOT}(Q)) = \text{ROOT}(D_c)$. $\text{ROOT}(D_c) = \text{SHADOW}(\text{ROOT}(Q))$, by the construction of D_c, and therefore $\psi(\text{ROOT}(Q)) = \text{ROOT}(Q)$.

2. Axis preservation: Let $u \in Q$, $u \neq \text{ROOT}(Q)$, and let $v = \psi(u) = \text{SHADOW}^{-1}(\phi(u))$ and $w = \psi(\text{PARENT}(u)) = \text{SHADOW}^{-1}(\phi(\text{PARENT}(u)))$.

If AXIS(u) = `child`, then by the axis match property of ϕ, $\phi(u)$ must be a child of ϕ(PARENT(u)). By the construction of D_c, a shadow node SHADOW(v') is a child of another shadow node SHADOW(w'), only if v' is a child of w' and AXIS(v') = `child`. Therefore, $v = \psi(u)$ must be a child of $w = \psi$(PARENT(u)) and have a `child` axis.

If AXIS(u) = `descendant`, then by the axis match property of ϕ, $\phi(u)$ must be a descendant of ϕ(PARENT(u)). By the construction of D_c, a shadow node SHADOW(v') is a descendant of another shadow node SHADOW(w'), only if v' is a descendant of w'. Therefore, $v = \psi(u)$ must be a descendant of $w = \psi$(PARENT(u)).

3. Node test preservation: Let $u \in Q$, and let $v = \psi(u) =$ SHADOW$^{-1}(\phi(u))$. Suppose NTEST(u) $\neq$ `*`. Therefore, by the node test match property of ϕ, NAME($\phi(u)$) = NTEST(u). In particular, the name of $\phi(u)$ is not the auxiliary name returned by the function `getAuxiliaryName(Q)`, implying that v does not have the wildcard node test. By the construction of D_c, since $\phi(u) =$ SHADOW(v), the name of $\phi(u)$ must equal NTEST(v). We conclude that NTEST($\psi(u)$) = NTEST(v) = NAME($\phi(u)$) = NTEST(u). $\qquad\square$

We can now prove that the canonical matching is unique:

Lemma 6.15. ϕ_c *is the only matching of D_c and Q.*

Proof. Suppose, to reach a contradiction, there exists a matching ϕ of D_c and Q, and $\phi \neq \phi_c$. Therefore, there is some node $v \in Q$, so that $\phi(v) \neq$ SHADOW(v). Any matching is also a structural matching. So, from Lemma 6.12, we know that $\phi(v) =$ SHADOW(u) for some node $u \in Q$, $u \neq v$.

Let $\psi(w) =$ SHADOW$^{-1}(\phi(w))$ be the structural query automorphism induced by ϕ (Lemma 6.14). We have: $\psi(v) = u$.

We first note that, without loss of generality, v is a leaf. Suppose not. Let v' be any child of v. We claim that also $\phi(v') \neq$ SHADOW(v'). If not, then $\psi(v') = v'$. By the axis preservation property of ψ, $\psi(v') = v'$ is a descendant of $\psi(v) = u$. We thus have: v' is a child of v and a descendant of u, and $v \neq u$. That must mean that v is a descendant of u. In particular, DEPTH(v) > DEPTH(u). This contradicts Proposition 6.10, because $u = \psi(v)$. Therefore, $\phi(v') \neq$ SHADOW(v'). Continuing this way inductively, we will reach a leaf v'' in the subtree Q_v, for which $\phi(v'') \neq$ SHADOW(v''). So from now on we assume v itself was a leaf to begin with.

We conclude that $\psi(v) = u$ and v is a leaf. By Lemma 6.9, this means that v is a leaf in the structural domination set of u. We finish off the proof by a case analysis:

Case 1: u is a leaf. By the construction of the document D_c, SHADOW(u) has a single text node child, whose text content is a value α, which belongs to TRUTH(u) but does not

belong to TRUTH(w), for any leaf w in the structural domination set of u. In particular, $\alpha \notin$ TRUTH(v). We conclude that STRVAL($\phi(v)$) = STRVAL(SHADOW(u)) = $\alpha \notin$ TRUTH(v). This contradicts the value match property of ϕ.

Case 2: u is an internal node. By the construction of the document D_c, SHADOW(u) has a text node child, preceding all its other children, whose text content α is not a prefix of any string in TRUTH(w), for all leaf nodes w in the structural domination set of u. In particular, α is not a prefix of any value in TRUTH(v). Note that α is a prefix of STRVAL(SHADOW(u)), and therefore STRVAL(SHADOW(u)) $\notin$ TRUTH(v). Again, this contradicts the value match property of ϕ.

We conclude that ϕ cannot be a valid matching, and thus ϕ_c is the only matching of D_c and Q. □

The following is a useful property of canonical documents (proof in Appendix E).

Proposition 6.16. *For any node $u \in Q$, no descendant of* SHADOW(u) *has a matching with u.*

Canonical documents and homomorphisms We show below how to translate the matching of the canonical document and the query into matchings of related documents and the query via the notion of homomorphisms discussed in Section 6.1.

Proposition 6.17. *Let ξ be a weak homomorphism from the canonical document D_c to a document D. Then, $\xi \circ \phi_c$ is a matching of D and Q.*

Proof. Follows immediately from Lemma 6.4 and the observation that the canonical matching is leaf-preserving. □

For the purpose of the next lemma, we need to introduce a class of homomorphisms:

Definition 6.18 (Internal node preserving homomorphism). A weak homomorphism ξ from a subtree D_x to a subtree $D'_{x'}$ is called *internal node preserving*, if for every internal node $y \in D_x$, the following hold: (1) $\xi(y)$ is an internal node; (2) if y has a `text` node child preceding its other children, then so does $\xi(y)$, and the text contents of the two `text` nodes are the same; and (3) if y does not have a `text` node child preceding its other children, then also $\xi(y)$ does not have one.

Lemma 6.19. *Let ξ be an internal node preserving weak homomorphism from a document D to D_c, and let ϕ be a matching of D and Q. Then, the mapping $\eta \stackrel{\text{def}}{=} \xi \circ \phi$ is a matching of D_c and Q (and thus equals the canonical matching ϕ_c).*

Note that this lemma does not follow directly from Lemma 6.4, because the matching ϕ is not guaranteed to be leaf-preserving. The proof appears in Appendix E.

7 Space lower bounds: full version

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 each of the lower bounds is as follows. Each lower bound is associated with a fragment $\mathcal{F}$ of XPath, to which it applies. In all three cases, this fragment is a subset of Redundancy-free XPath (see Section 5). We fix an arbitrary query $Q \in \mathcal{F}$, and prove a lower bound on the data complexity of BOOLEVAL$_Q$ (recall definition from Section 3.1). 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.

7.1 Query frontier size

In this section we extend the query frontier size lower bound (Theorem 4.2) to arbitrary redundancy-free queries.

Theorem 7.1. *Let Q be a redundancy-free query. Then, for every streaming algorithm that computes BOOLEVAL$_Q$, there is at least one document on which the algorithm requires at least FS(Q) bits of space.*

Proof. We create from the function BOOLEVAL$_Q$ a two-argument function BOOLEVAL$_Q^2$ as described in Section 3.2: the first argument is a prefix of an XML stream and the second argument is a suffix of an XML stream. We will describe a family of documents w.r.t. which there is a FS(Q) lower bound on the communication complexity of BOOLEVAL$_Q^2$. It would follow (Lemma 3.7) that any streaming algorithm evaluating BOOLEVAL$_Q$ needs to use at least FS(Q) bits of space on at least one of the documents in the family.

Let $D = D_c$ be the canonical document corresponding to Q (see Section 6.4). 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). It thus suffices to show a FS(D) lower bound on the communication complexity of BOOLEVAL$_Q^2$.

We use the fooling set technique (see Section 3.2) to prove the lower bound for BOOLEVAL_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 weakly homomorphic to the canonical document D.

Let x be the node in D with the largest frontier. Without loss of generality, x is a shadow node (because any artificial node has a descendant which is a shadow node and whose frontier is at least as large). We associate with each subset T of $\text{F}(x)$ (the frontier at 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 XML stream segments defined as follows. Let $x_1, \ldots, x_\ell$ be the nodes in $\text{PATH}(x)$ (that is, $x_1 = \text{ROOT}(D)$ and $x_\ell = x$). Recall 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 $\ell - 1$ segments: $\alpha_T = \alpha_{T,1} \circ \cdots \circ \alpha_{T,\ell-1}$ 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 $\mathtt{a}_i = \text{NAME}(x_i)$, 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 $\text{F}(x) \setminus T$. Then, $\alpha_{T,i}$ is defined as: $\langle \mathtt{a}_i \rangle \circ D_{y_1} \circ \cdots \circ D_{y_k}$, where D_{y_j} is the XML stream segment representing the subtree of D rooted at y_j. Similarly, $\beta_{T,i} = D_{z_1} \circ \cdots \circ D_{z_m} \circ \langle /\mathtt{a}_i \rangle$.

Example. Consider the query $Q = \mathtt{/a[c[.//e\ and\ f]\ and\ b > 5]}$. The corresponding canonical document is the following:

$$D = \langle \mathtt{a} \rangle\ \langle \mathtt{c} \rangle\ \langle \mathtt{Z} \rangle\langle \mathtt{e/} \rangle\langle \mathtt{/Z} \rangle\ \langle \mathtt{f/} \rangle\ \langle \mathtt{/c} \rangle\ \langle \mathtt{b} \rangle 6\langle \mathtt{/b} \rangle\ \langle \mathtt{/a} \rangle.$$

The largest frontier of this document is the frontier at the "e" node, which consists of the nodes $\{\ \mathtt{e},\ \mathtt{f},\ \mathtt{b}\ \}$. Consider the subset of the frontier $T = \{\mathtt{b}, \mathtt{f}\}$. Then, the values of α_T and β_T in this case are the following:

$$\alpha_T = \langle \mathtt{a} \rangle\ \langle \mathtt{b} \rangle 6\langle \mathtt{/b} \rangle\ \langle \mathtt{c} \rangle\langle \mathtt{f/} \rangle\langle \mathtt{Z} \rangle \qquad \beta_T = \langle \mathtt{e/} \rangle\langle \mathtt{/Z} \rangle\ \langle \mathtt{/c} \rangle\ \langle \mathtt{/a} \rangle.$$

Let D_T be the XML document represented by the stream $\alpha_T \circ \beta_T$. For two different subsets $T \neq T'$ of $\text{F}(x)$, let $D_{T,T'}$ be the document represented by the stream $\alpha_T \circ \beta_{T'}$, and let $D_{T',T}$ be the document represented by the stream $\alpha_{T'} \circ \beta_T$. The two following claims establish that S is indeed a fooling set, completing the proof of the theorem.

Claim 7.2. *For every T, D_T is a well-formed document and matches Q.*

Proof. It is easy to verify that D_T is identical to D, except that the children of each node $x_i \in \text{PATH}(x)$ are ordered as follows: first all the children that belong to T, then x_{i+1}, and then the children that belong to $\text{F}(x) \setminus T$. It is thus immediate to see that the mapping ξ that maps each node of D to its "copy" in D_T is a weak homomorphism. The claim now follows from Proposition 6.17. $\square$

Claim 7.3. *For every two distinct subsets T, T', at least one of $D_{T,T'}, D_{T',T}$ is well-formed and does not match Q.*

Proof. 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 $T \cup (\mathrm{F}(x) \setminus T')$ is a proper subset of $\mathrm{F}(x)$. Hence, there is a node $z \in \mathrm{F}(x)$, which does not belong to $T \cup (\mathrm{F}(x) \setminus T')$. Note that z cannot be an artificial node, because no artificial node belongs to the frontier of a shadow node.

It is easy to see that $D_{T,T'}$ and $D_{T',T}$ are well-formed documents, since the proper nesting of elements is maintained in both. $D_{T,T'}$ is identical to D, except for the following differences: for each $i = 1, \ldots, \ell$, the children of x_i that have copies in $D_{T,T'}$ are x_{i+1} (if $i < \ell$) and the children that belong to the set $T \cup (\mathrm{F}(x) \setminus T')$. The internal order among these children is not preserved in $D_{T,T'}$. Furthermore, if some child y of x_i ($y \neq x_{i+1}$) belongs both to T and to $\mathrm{F}(x) \setminus T'$, then it has two copies in $D_{T,T'}$. Nevertheless, since every node of $D_{T,T'}$ originates from a node of D, there is a natural mapping ξ that maps each node $D_{T,T'}$ to its "origin" in D. It is easy to verify that ξ is an internal node preserving weak homomorphism (recall Definition 6.18).

Suppose, to reach a contradiction, there exists a matching ϕ of $D_{T,T'}$ with Q. By Lemma 6.19, the map $\eta \overset{\text{def}}{=} \xi \circ \phi$ is a matching of D_c with Q and therefore equals the canonical matching ϕ_c. Every shadow node has a pre-image under ϕ_c. Therefore, every shadow node has to have a pre-image under ξ as well. However, z is a shadow node and is not in the image of ξ (because z has no "copy" in $D_{T,T'}$). We reached a contradiction. $\square$

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 3.9) to the function $\textsc{booleval}^2_Q$ and by the reduction lemma (Lemma 3.7). $\square$

7.2 Recursion depth

In this section we extend the recursion depth lower bound (Theorem 4.5) to arbitrary redundancy-free queries that contain the query `//a[b and c]` as a sub-query. Thus, the queries Q to which the lower bound below applies are ones in the following fragment of XPath:

Recursive XPath *Recursive XPath* is a subset of Redundancy-free XPath that consists of queries Q, which possess at least one node v with the following properties: (1) Either v or one of its ancestors has a descendant axis; and (2) v has at least two children with a child axis.

Remark. Ideally, one would want to prove the recursion depth lower bound for any query that consists of a node with a descendant axis. Yet, this is simply not true. For example, the query `//a` can be evaluated with only 1 bit of memory, regardless of the document's recursion depth, and the query `//a//b` can be evaluated with space proportional to the logarithm of the recursion depth. We were able to prove the lower bound only for queries that have at least one node with a descendant axis that has at least two sibling descendants with a child axis. The query `//a[b and c]` is a classical example of such a query. We believe the lower bound holds also for queries that have a node with a descendant axis that has a descendant with a child axis, e.g., `//a//b`. It is left open to extend our proof to such queries as well.

Theorem 7.4. *Let Q be any query in Recursive XPath, and let v be the node of Q, as defined above. Then, for any streaming algorithm that computes* BOOLEVAL$_Q$, *and for any integer $r \geq 1$, there is at least one document of recursion depth at most r w.r.t. v, on which the algorithm requires $\Omega(r)$ bits of space.*

Proof. As in the proof of Theorem 4.5, we use a reduction from the set disjointness problem. Recall that this problem has an $\Omega(r)$ communication lower bound. We will prove that given a streaming algorithm that computes BOOLEVAL$_Q$, and given any integer $r \geq 1$, if the algorithm uses at most C bits of space on any document of recursion depth at most r w.r.t. v, then we can design a communication protocol that solves the set disjointness problem with C bits of communication. It would then immediately follow that C has to be at least $\Omega(r)$.

If v has a descendant axis itself, define $v_1 = v$. Otherwise, let v_1 be the lowest ancestor of v with a descendant axis. Let $v_1, \ldots, v_k$ be the nodes along the path from v_1 to v (i.e., $v_k = v$). Note that $v_2, \ldots, v_k$ must have a child axis. Let v_0 denote the parent of v_1. Finally, let w_1, w_2 be the two children of $v_k = v$ with a child axis. See Figure 10 for a schematic illustration of the query tree.

Example. Suppose $Q = $ `//d[f and a[b and c]]` (see Figure 11). Here, $k = 2$, v_0 is the root of the query, v_1 is the node named "`d`", v_2 is the node named "`a`", w_1 is the node named "`b`", and w_2 is the node named "`c`".

Let $D = D_c$ be the canonical document corresponding to the query Q (see Section 6.4), and let $\phi = \phi_c$ be the canonical matching of D and Q. Recall that for each node $u \in Q$, $\phi(u) = $ SHADOW(u). Also recall that if u has a descendant axis, then $\phi(u)$ is a descendant of $\phi(\text{PARENT}(u))$, following a chain of $h + 1$ "artificial" nodes, where h is the longest chain of wildcard nodes in Q. In our case, v_1 has a descendant axis; so let y denote the child of $\phi(v_0) = \phi(\text{PARENT}(v_1))$, which is the first artificial node in the chain of $h + 1$ artificial nodes preceding $\phi(v_1)$. See Figure 12 for a schematic illustration of the canonical document.
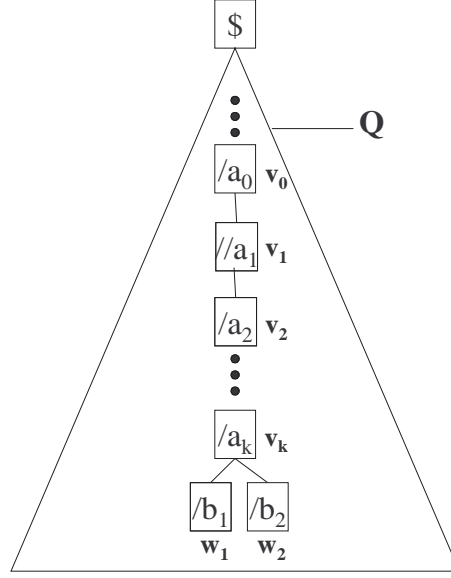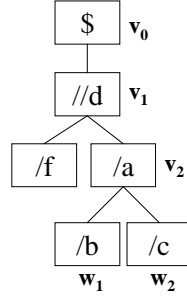
41

Figure 10: A query in Recursive XPath

Figure 11: The query `//d[f and a[b and c]]`

Consider the stream representation of the document D, and split it into seven contiguous segments as follows:

1. γ_{prefix} is the prefix of the stream ending just before the `startElement` event of the element y.

2. $\gamma_{\text{y-beg}}$ is the segment starting with the `startElement` event of y and ending just before the `startElement` event of $\phi(w_1)$.

3. γ_{w_1} is the segment containing the element $\phi(w_1)$.

4. $\gamma_{\text{y-mid}}$ is the segment starting after the `endElement` event of $\phi(w_1)$ and ending just before the `startElement` event of $\phi(w_2)$.
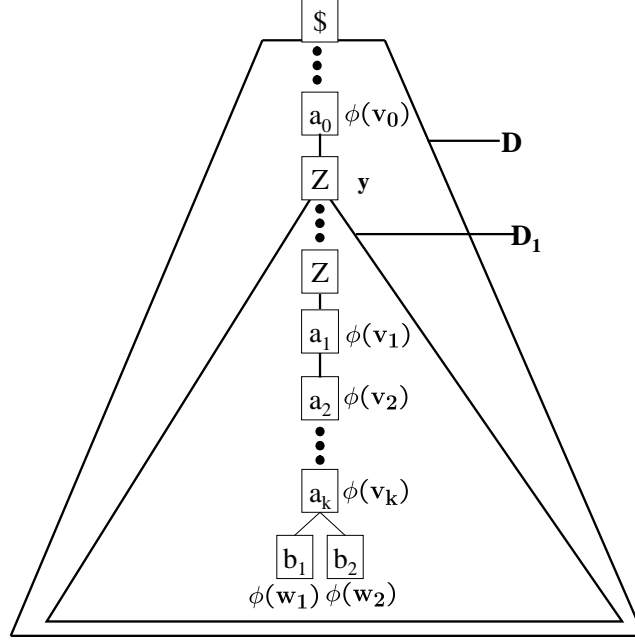
42

Figure 12: A canonical document corresponding to a query in Recursive XPath

5. γ_{w_2} is the segment containing the element $\phi(w_2)$.

6. $\gamma_{\text{y-end}}$ is the segment following the endElement event of $\phi(w_2)$ and ending with the endElement event of y.

7. γ_{suffix} is the rest of the XML stream.

Example. The following is the canonical document corresponding to the example query from above and the corresponding partition into segments. Z is the "auxiliary name". (See also Figure 13.)

No.	Event	No.	Event
0)	$\langle \$ \rangle$	7)	$\langle /\text{a} \rangle$
1)	$\langle \text{Z} \rangle$	8)	$\langle /\text{d} \rangle$
2)	$\langle \text{d} \rangle$	9)	$\langle /\text{Z} \rangle$
3)	$\langle \text{f}/ \rangle$	10)	$\langle /\$ \rangle$
4)	$\langle \text{a} \rangle$		
5)	$\langle \text{b}/ \rangle$		
6)	$\langle \text{c}/ \rangle$		

Segment	Event range
γ_{prefix}	$[0]$
$\gamma_{\text{y-beg}}$	$[1\text{-}4]$
γ_{w_1}	$[5]$
$\gamma_{\text{y-mid}}$	$[\ ]$
γ_{w_2}	$[6]$
$\gamma_{\text{y-end}}$	$[7\text{-}9]$
γ_{suffix}	$[10]$

We next describe how to translate an input $(\mathbf{s}, \mathbf{t})$ of the set disjointness problem into an XML document $D_{\mathbf{s},\mathbf{t}}$ of recursion depth at most r w.r.t. v_k. Any $\mathbf{s} \in \{0,1\}^r$ is translated into

$ $\phi(\mathbf{v_0})$

Z $\mathbf{y}$

d $\phi(\mathbf{v_1})$

f   a $\phi(\mathbf{v_2})$

b   c

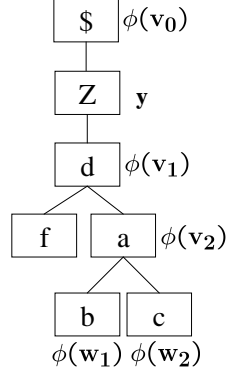$\phi(\mathbf{w_1})$ $\phi(\mathbf{w_2})$

Figure 13: The canonical document corresponding to the query `//d[f and a[b and c]]`

a prefix of an XML stream $\alpha = \gamma_{\text{prefix}} \circ \alpha_1 \circ \cdots \circ \alpha_r$, where $\alpha_i = \gamma_{\text{y-beg}} \circ \gamma_{w_1} \circ \gamma_{\text{y-mid}}$, if $\mathbf{s}_i = 1$, and $\alpha_i = \gamma_{\text{y-beg}} \circ \gamma_{\text{y-mid}}$, if $\mathbf{s}_i = 0$. That is, α_i includes a copy of the subtree rooted at $\phi(w_1)$, if only if $\mathbf{s}_i = 1$. Similarly, any $\mathbf{t} \in \{0,1\}^r$ is translated into a suffix of an XML stream $\beta = \beta_r \circ \cdots \circ \beta_1 \circ \gamma_{\text{suffix}}$, where $\beta_i = \gamma_{w_2} \circ \gamma_{\text{y-end}}$, if $\mathbf{t}_i = 1$, and $\beta_i = \gamma_{\text{y-end}}$, if $\mathbf{t}_i = 0$. That is, β_i includes a copy of the subtree rooted at $\phi(w_2)$, if only if $\mathbf{t}_i = 1$. $D_{\mathbf{s},\mathbf{t}}$ is the document obtained by concatenating α and β.

Example. Consider our example query from above, and suppose $r = 3$, $\mathbf{s} = 110$, and $\mathbf{t} = 010$. Then, the stream representing the document $D_{\mathbf{s},\mathbf{t}}$ is as follows (see also Figure 14):

Segment	No.	Event	Segment	No.	Event	Segment	No.	Event
γ_{prefix}	0)	$\langle \$ \rangle$		9)	$\langle \text{a} \rangle$	β_2	**18)**	$\langle \boldsymbol{c/} \rangle$
α_1	1)	$\langle \text{Z} \rangle$		**10)**	$\langle \boldsymbol{b/} \rangle$		19)	$\langle \text{/a} \rangle$
	2)	$\langle \text{d} \rangle$	α_3	11)	$\langle \text{Z} \rangle$		20)	$\langle \text{/d} \rangle$
	3)	$\langle \text{f/} \rangle$		12)	$\langle \text{d} \rangle$		21)	$\langle \text{/Z} \rangle$
	4)	$\langle \text{a} \rangle$		13)	$\langle \text{f/} \rangle$	β_1	22)	$\langle \text{/a} \rangle$
	5)	$\langle \boldsymbol{b/} \rangle$		14)	$\langle \text{a} \rangle$		23)	$\langle \text{/d} \rangle$
α_2	6)	$\langle \text{Z} \rangle$	β_3	15)	$\langle \text{/a} \rangle$		24)	$\langle \text{/Z} \rangle$
	7)	$\langle \text{d} \rangle$		16)	$\langle \text{/d} \rangle$	γ_{suffix}	25)	$\langle \text{/\$} \rangle$
	8)	$\langle \text{f/} \rangle$		17)	$\langle \text{/Z} \rangle$			

$D_{\mathbf{s},\mathbf{t}}$ is well formed, because nesting of elements is properly maintained. We next describe the exact structure of $D_{\mathbf{s},\mathbf{t}}$. Refer to Figure 15 for assistance.

Let D_1 be the subtree of D rooted at y, and let D_0 be the document D after removing D_1. $D_{\mathbf{s},\mathbf{t}}$ is the same as D, except that the subtree D_1 is replaced by a new subtree E_1, which we describe below. We denote by E_0 the document $D_{\mathbf{s},\mathbf{t}}$ after removing E_1. Clearly, there is an isomorphism f_0 from E_0 to D_0. Let $g_0 = f_0^{-1}$ be the inverse isomorphism.
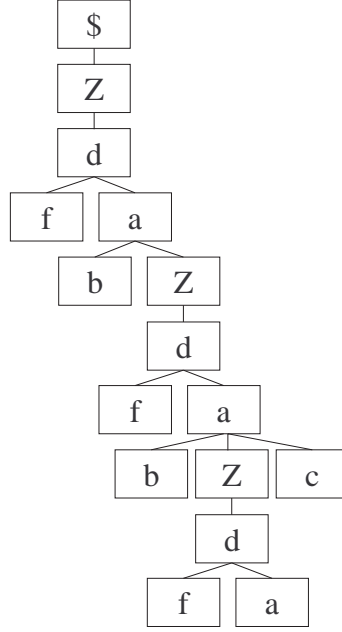
Figure 14: The document $D_{110,010}$ corresponding to the query `//d[f and a[b and c]]`

For each of the four bit pairs $(b_1, b_2) \in \{00, 01, 10, 11\}$, define the tree G_{b_1,b_2} to be the same as D_1, except that: (1) the subtree rooted at $\phi(w_1)$ is excluded from G_{b_1,b_2} if and only if $b_1 = 0$; and (2) the subtree rooted at $\phi(w_2)$ is excluded from G_{b_1,b_2} if and only if $b_2 = 0$. Note that G_{b_1,b_2} can be embedded in D_1, and therefore there exists an injective weak homomorphism f_{b_1,b_2} from G_{b_1,b_2} to D_1. The only nodes of D_1 to be excluded from the image of f_{b_1,b_2} are nodes in the subtrees rooted at $\phi(w_1)$ (if $b_1 = 0$) and at $\phi(w_2)$ (if $b_2 = 0$). We will denote the inverse mapping from the image of f_{b_1,b_2} to D_1 by g_{b_1,b_2}.

For each $i = 1, \ldots, r$, define $F_i \overset{\text{def}}{=} G_{\mathbf{s}_i,\mathbf{t}_i}$, $f_i \overset{\text{def}}{=} f_{\mathbf{s}_i,\mathbf{t}_i}$, and $g_i \overset{\text{def}}{=} g_{\mathbf{s}_i,\mathbf{t}_i}$. Also inductively define subtrees $E_r, \ldots, E_1$ as follows: $E_r = F_r$; assuming that E_{i+1} is defined, E_i is the same as F_i, except that the root of E_{i+1} is attached as a child of $g_i(\phi(v_k))$. E_1 is the subtree that replaces D_1 in $D_{\mathbf{s},\mathbf{t}}$.

Finally, define a mapping f from $D_{\mathbf{s},\mathbf{t}}$ to D, as follows:

$$f(x) = \begin{cases} f_0(x) & \text{if } x \in E_0 \\ f_i(x) & \text{if } x \in F_i \end{cases} .$$

We note that f is well-defined, but is not even a structural homomorphism from $D_{\mathbf{s},\mathbf{t}}$ to D, because for each $i = 2, \ldots, r$, the root of F_i (i.e., $g_i(y)$) is a child of $g_{i-1}(\phi(v_k))$, yet $f(g_i(y)) = y$ is an ancestor of $f(g_{i-1}(\phi(v_k))) = \phi(v_k)$.

Let $I = \{i \in \{1, \ldots, r\} : \mathbf{s}_i = \mathbf{t}_i = 1\}$. We will prove two crucial facts about the
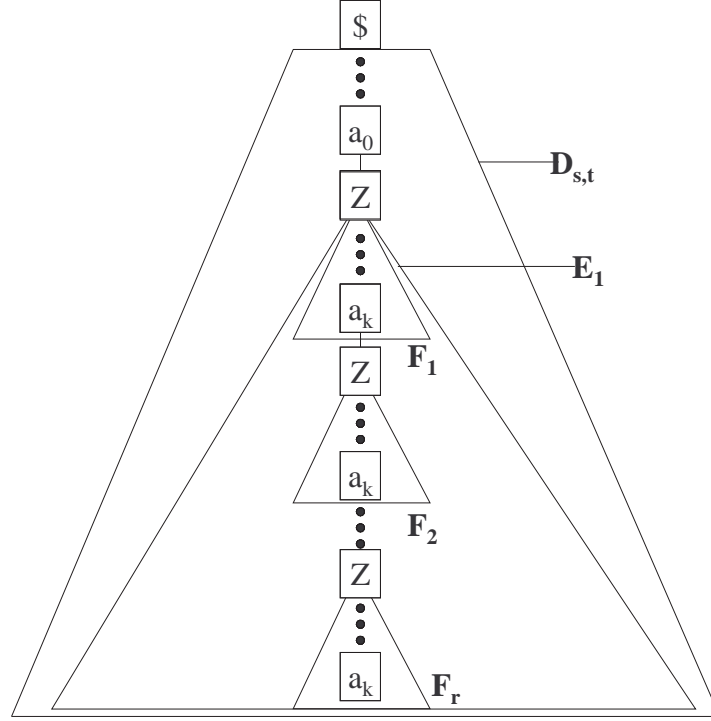
45

Figure 15: A schematic illustration of the document $D_{\mathbf{s,t}}$

document $D_{\mathbf{s,t}}$:

Lemma 7.5. *If $I \neq \emptyset$, then $D_{\mathbf{s,t}}$ matches Q.*

Lemma 7.6. *The only nodes in $D_{\mathbf{s,t}}$ that can match v_k are nodes in the set $\{g_i(\phi(v_k)) \; : \; i \in I\}$. In particular, if $I = \emptyset$, then $D_{\mathbf{s,t}}$ does not match Q.*

Before we prove the two lemmas, let us see how to use them to finish off the proof of the theorem. By Lemma 7.6, only the nodes $g_1(\phi(v_k)), \ldots, g_r(\phi(v_k))$ can potentially match v_k. This immediately implies that the recursion depth of $D_{\mathbf{s,t}}$ w.r.t. v_k can be at most r.

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 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.

Since the document $D_{\mathbf{s,t}}$ is of recursion depth at most r w.r.t. $v_k = v$, the state of the algorithm requires at most C bits to describe, and hence the protocol uses at most C bits of communication. We next prove that it computes the function DISJ correctly. Suppose,

initially, that $S \cap T \neq \emptyset$. Then, there exists some index $1 \leq i \leq r$, such that both $\mathbf{s}_i$ and $\mathbf{t}_i$ are 1. By Lemma 7.5, this means that $D_{\mathbf{s},\mathbf{t}}$ matches Q. Hence, the protocol will indeed declare S and T as intersecting. Suppose now that $S \cap T = \emptyset$. Then, for all $1 \leq i \leq r$, $\mathbf{s}_i = 0$ or $\mathbf{t}_i = 0$. It follows from Lemma 7.6 that $D_{\mathbf{s},\mathbf{t}}$ does not match Q, and therefore the protocol will declare the sets as disjoint. □

For brevity, we denote $E = D_{\mathbf{s},\mathbf{t}}$. $g_1(\phi(v_1)), \ldots, g_1(\phi(v_k)), \ldots, g_r(\phi(v_1)), \ldots, g_r(\phi(v_k))$ lie on the same root-to-leaf path in E. We will call this path the "spine" of E. The following are easy to verify from the construction of E:

Observation 7.7. *For any node $x \in E$, which does not belong to the spine, f restricted to E_x (the subtree of E rooted at x) is an isomorphism of E_x with $D_{f(x)}$ (the subtree of D rooted at $f(x)$).*

Observation 7.8. *If $i \in I$, then g_i is an isomorphism of D_1 with F_i.*

Proof of Lemma 7.5. Suppose there exists some $1 \leq i \leq r$, such that $\mathbf{s}_i = \mathbf{t}_i = 1$. By Lemma 5.10, it suffices to exhibit a matching μ of Q and E. To this end, we will construct μ as a hybrid matching (see Section 6.2).

The canonical matching ϕ matches D with Q. Let Q_0 be the query Q after removing the subtree rooted at v_1. Let ϕ_0 be the restriction of ϕ to Q_0. Since ϕ is a leaf-preserving matching of D and Q, it is immediate to check that ϕ_0 is a leaf-preserving matching of D_0 and Q_0. Since there is a weak homomorphism g_0 from D_0 to E_0, then $\mu_0 \stackrel{\text{def}}{=} g_0 \circ \phi_0$ is a matching of E_0 and Q_0 (Lemma 6.4).

Let Q_1 be the subtree of Q rooted at v_1, and let ϕ_1 be the restriction of ϕ to Q_1. Let D_1' be the subtree of D rooted at $\phi(v_1)$. Recall that D_1 is the subtree of D rooted at y, which is the artificial child of $\phi(v_0)$. y is an ancestor of $\phi(v_1)$, and therefore D_1' is a subtree of D_1.

Again, it is immediate to check that ϕ_1 is a leaf-preserving matching of D_1' and Q_1. By Observation 7.8 above, g_i is a isomorphism from D_1 to F_i. Since F_i embeds in E_i, g_i is also a weak homomorphism from D_1 to E_i. Let E_i' be the subtree of E_i rooted at $g_i(\phi(v_1))$. We obtain that the restriction of g_i to D_1', which we denote by g_i', is a weak homomorphism from D_1' to E_i'. Therefore, $\mu_1 \stackrel{\text{def}}{=} g_i' \circ \phi_1$ is a matching of E_1' with Q_1 (Lemma 6.4).

We define μ to be the hybrid mapping induced by μ_0 and μ_1. In order for μ to be a matching, we need $\mu_1(v_1)$ to relate to $\mu_0(\text{PARENT}(v_1))$ according to $\text{AXIS}(v_1)$ (recall Lemma 6.7). Recall that v_1 has a `descendant` axis. So we need to prove that $\mu_1(v_1) = g_i(\phi(v_1))$ is a descendant of $\mu_0(\text{PARENT}(v_1)) = g_0(\phi(v_0))$. By definition, $g_i(\phi(v_1)) \in E_i \subseteq E_1$ and all the nodes in E_1 are descendants of $g_0(\phi(v_0))$. So in particular $g_i(\phi(v_1))$ is a descendant of $g_0(\phi(v_0))$. □

47

Proof of Lemma 7.6. Let μ be any matching of E and Q. We will show that $\mu(v_k)$ has to equal $g_i(\phi(v_k))$ for some $i \in I$.

Recall that $v_k = v$ has two children w_1, w_2 with a child axis. We want to show that $\mu(w_1) = g_i(\phi(w_1))$ for some $i \in \{1, \ldots, r\}$ and that $\mu(w_2) = g_j(\phi(w_2))$ for some $j \in \{1, \ldots, r\}$. We will prove the former; the proof of the latter is identical.

Claim 7.9. $\mu(w_1) = g_i(\phi(w_1))$ *for some* $i \in \{1, \ldots, r\}$.

Before we show the proof of this claim, let us use it to complete the proof of the lemma. Let i be so that $\mu(w_1) = g_i(\phi(w_1))$ and let j be so that $\mu(w_2) = g_j(\phi(w_2))$. In particular, g_i is defined on $\phi(w_1)$ and g_j is defined on $\phi(w_2)$. w_1, w_2 are children of v_k and have a child axis. Therefore, by the axis match property of μ, $\mu(w_1), \mu(w_2)$ are children of $\mu(v_k)$. This means that $\mu(v_k) = \text{PARENT}(g_i(\phi(w_1))) = g_i(\phi(v_k))$ and $\mu(v_k) = \text{PARENT}(g_j(\phi(w_2))) = g_j(\phi(v_k))$. In other words, $i = j$, and $\mu(v_k) = g_i(v_k)$. Since g_i is defined on both $\phi(w_1)$ and $\phi(w_2)$, $i \in I$. $\square$

In order to prove Claim 7.9 we need a few preliminaries. The nodes of E, like the nodes of D, can be classified as "shadow nodes" or "artificial nodes". A node $x \in E$ is artificial if and only if $f(x)$ is an artificial node of D. The same argument as in the proof of Lemma 6.12 shows that there are no artificial nodes in the image of μ:

Claim 7.10. *For all* $u \in Q$, $\mu(u)$ *is not an artificial node.*

Define $v_{k+1} = w_1$. Note that the nodes $v_1, \ldots, v_{k+1}$ form a path segment in Q, v_1 has a `descendant` axis, and $v_2, \ldots, v_{k+1}$ have a `child` axis. We show next that if μ maps any of these nodes to a node in F_i, then it maps all the others to nodes in F_i as well:

Claim 7.11. *For each* $i \in \{1, \ldots, r\}$, *if* $\mu(v_{k+1}) \in F_i$, *then also* $\mu(v_1) \in F_i$.

Proof. Since $v_1, \ldots, v_{k+1}$ form a path segment in Q, and since $v_2, \ldots, v_{k+1}$ all have a `child` axis, then by the axis match property of μ, also the sequence $\mu(v_1), \ldots, \mu(v_{k+1})$ forms a path segment in E.

Recall that the root of F_i is the artificial node $g_i(y)$, which is followed by a path of h additional artificial nodes $g_i(y_1), \ldots, g_i(y_h)$. The rest of the nodes in F_i are all descendants of $g_i(y_h)$. Since $\mu(v_{k+1})$ belongs to F_i and is not an artificial node, it must be a descendant of $g_i(y_h)$. If $\mu(v_1) \notin F_i$, then $\mu(v_1)$ has to be an ancestor of $g_i(y)$. This implies that $g_i(y), g_i(y_1), \ldots, g_i(y_h)$ all lie on the path segment $\mu(v_1), \ldots, \mu(v_{k+1})$. This is impossible, because as mentioned earlier, there are no artificial nodes in the image of μ. $\square$

Let Q_0 be the query Q after removing the subtree rooted at v_1. Let Q_1 be the subtree of Q rooted at v_1. Let $u_1, \ldots, u_t$ be the path from the root of Q to v_{k+1} (that is, $u_1 = \text{ROOT}(Q)$ and $u_t = v_{k+1}$). We prove the following:

Claim 7.12. *There exists a matching η of E with Q that agrees with μ on Q_1 and that sat- isfies the following: for all $j = 2, \ldots, t$, $f(\eta(u_j))$ relates to $f(\eta(u_{j-1}))$ according to $\mathrm{AXIS}(u_j)$.*

Proof. There are two cases: either $\mu(v_{k+1}) \in E_0$ or $\mu(v_{k+1}) \in E_1$. If $\mu(v_{k+1}) \in E_0$, then also its ancestors $\mu(u_1), \ldots, \mu(u_{t-1})$ belong to E_0. Recall that f restricted to E_0 (a.k.a. f_0) is an isomorphism. We can therefore choose $\eta = \mu$, and the desired property stated in the claim follows from the axis match property of μ.

Consider then the case that $\mu(v_{k+1}) \in E_1$. By Claim 7.11, $\mu(v_1)$ also belongs to E_1. We next prove that, without loss of generality, μ agrees with $g_0 \circ \phi$ on Q_0 (Q_0 is the query Q after removing the subtree rooted at v_1).

Let ϕ_0 be the restriction of ϕ to Q_0. It is easy to verify that ϕ_0 is a leaf-preserving matching of D_0 and Q_0. Since g_0 is a weak homomorphism from D_0 to E_0, then $\eta_0 \overset{\text{def}}{=} g_0 \circ \phi_0$ is a matching of E_0 with Q_0 (Lemma 6.4).

Let η_1 be the restriction of μ to Q_1 (Q_1 is the subtree of Q rooted at v_1). Clearly, η_1 is a matching of $\mu(v_1)$ with v_1. $\mu(v_1)$ belongs to E_1 and all the nodes in E_1 are descendants of $g_0(\phi(v_0))$. Therefore, $\eta_1(v_1) = \mu(v_1)$ relates to $\eta_0(\mathrm{PARENT}(v_1)) = g_0(\phi(v_0))$ according to $\mathrm{AXIS}(v_1)$ (which is $\mathtt{descendant}$). Hence, the hybrid mapping η induced by η_0 and η_1 is a matching of E with Q (Lemma 6.7). Note that η agrees with μ on Q_1 and with $g_0 \circ \phi$ on Q_0.

Let i be the one for which $\mu(v_1) \in F_i$. Since, $\mu(v_1), \ldots, \mu(v_{k+1})$ all belong to F_i (Claim 7.11), then $f(\mu(v_j)) = f_i(\mu(v_j))$ for all $j = 1, \ldots, k+1$. Recall that f_i is an injective weak homomorphism from F_i to D_1. Thus, the property stated in the claim holds for $f(\mu(v_2)), \ldots, f(\mu(v_{k+1}))$, due to the axis match property of μ and the axis preservation property of f_i. Since η agrees with μ on Q_1 and $v_1, \ldots, v_{k+1} \in Q_1$, then this property also holds for $f(\eta(v_2)), \ldots, f(\eta(v_{k+1}))$.

Recall that $v_2, \ldots, v_{k+1}$ are the last k nodes on the path $u_1, \ldots, u_t$, so we are left to address only the first $t - k$ nodes among $f(\eta(u_1)), \ldots, f(\eta(u_t))$.

Since η agrees with $g_0 \circ \phi$ on Q_0, then for all $j = 1, \ldots, t - k - 1$, $f(\eta(u_j)) = \phi(u_j)$. Thus, the property stated in the claim holds for these nodes due to the axis match property of ϕ.

The last missing component is to show that $f(\eta(u_{t-k})) = f(\eta(v_1))$ relates to $f(\eta(u_{t-k-1})) = f(\eta(v_0))$ according to $\mathrm{AXIS}(v_1)$. v_1 has a $\mathtt{descendant}$ axis, so we need to show that $f(\eta(v_1))$ is a descendant of $f(\eta(v_0))$. Since $\eta(v_1) = \mu(v_1) \in E_1$, then $f(\eta(v_1)) \in D_1$, and is therefore a descendant of $\phi(v_0)$ (recall that the root of D_1 is y, which is a child of $\phi(v_0)$). Since η agrees with $g_0 \circ \phi$ on Q_0, then $f(\eta(v_0)) = \phi(v_0)$. Therefore $f(\eta(v_1))$ is indeed a descendant of $f(\eta(v_0))$ as desired. $\qquad\square$

Proof of Claim 7.9. Suppose, to reach a contradiction, that $\mu(w_1) \neq g_i(\phi(w_1))$ for all $i = 1, \ldots, r$. This means that $f(\mu(w_1)) \neq \phi(w_1)$. Let η be the matching guaranteed by Claim 7.12. Since η agrees with μ on Q_1 and since $w_1 \in Q_1$, then also $f(\eta(w_1)) \neq \phi(w_1)$.

Consider the path $u_1, \ldots, u_t$ from the root of Q to w_1. Since $f(\eta(u_1)) = \phi(u_1)$ but $f(\eta(u_t)) \neq \phi(u_t)$, there is some $2 \leq j \leq t$ which is the first to satisfy $f(\eta(u_j)) \neq \phi(u_j)$. There are now two cases: either $\eta(u_j)$ belongs to the spine of E or not. We will start by analyzing the latter case.

Let Q_1 be the subtree of Q rooted at u_j, and let η_1 be the restriction of η to Q_1. Since η is a matching of E with Q, then in particular η_1 is a matching of $\eta(u_j)$ with u_j. Let f' denote the restriction of f to $E_{\eta(u_j)}$. By Observation 7.7, since $\eta(u_j)$ does not belong to the spine, f' is an isomorphism of $E_{\eta(u_j)}$ with $D_{f(\eta(u_j))}$. It follows that $\rho_1 \overset{\text{def}}{=} f' \circ \eta_1$ is a matching of $f(\eta(u_j))$ with u_j (Lemma 6.2).

Let Q_0 denote the query Q with the subtree Q_1 removed. The restriction of the canonical matching ϕ to Q_0 induces a matching ρ_0 of D with Q_0.

From Claim 7.12, we know that $f(\eta(u_j))$ relates to $f(\eta(u_{j-1}))$ according to $\text{AXIS}(u_j)$. However, by the choice of j, $f(\eta(u_{j-1})) = \phi(u_{j-1})$. We thus got a node in D (namely, $f(\eta(u_j))$) that has a matching with u_j and relates to $\phi(\text{PARENT}(u_j))$ according to $\text{AXIS}(u_j)$. Therefore, by Lemma 6.7, the hybrid mapping ρ induced by ρ_0 and ρ_1 is matching of D with Q. ρ cannot equal the canonical matching, because $\rho(u_j) = f(\eta(u_j)) \neq \phi(u_j)$. This contradicts the uniqueness of the canonical matching (Lemma 6.15).

So let's move on to the other case: $\eta(u_j)$ belongs to the spine. We note that for every node x in the spine of E, $f(x)$ is a node on the root-to-leaf path in D that goes through $\phi(u_1), \ldots, \phi(u_t)$. In particular, $f(\eta(u_j))$ belongs to this path. Since $f(\eta(u_j)) \neq \phi(u_j)$, it is either an ancestor of $\phi(u_j)$ or a descendant of $\phi(u_j)$.

We first exclude the possibility that $f(\eta(u_j))$ is an ancestor of $\phi(u_j)$. By Claim 7.12, $f(\eta(u_1)), \ldots, f(\eta(u_j))$ lie on a single root-to-leaf path in D in this order. Therefore, all these nodes, and not only $f(\eta(u_j))$ belong to the path that goes through $\phi(u_1), \ldots, \phi(u_j)$. By Claim 7.10, none of the nodes $f(\eta(u_1)), \ldots, f(\eta(u_j))$ is artificial (note that the claim is stated for μ but in fact holds for any matching of E with Q, and in particular for η). Thus, if $f(\eta(u_j))$ is an ancestor of $\phi(u_j)$, then $\phi(u_j)$ has at least j non-artificial ancestors. However, $\phi(u_j)$ has exactly $j - 1$ non-artificial ancestors, namely $\phi(u_1), \ldots, \phi(u_{j-1})$. It follows that $f(\eta(u_j))$ cannot be an ancestor of $\phi(u_j)$.

So assume that $f(\eta(u_j))$ is a descendant of $\phi(u_j)$. We will need the following claim:

Claim 7.13. *Let $j \leq s \leq t$. If for all $j \leq \ell < s$, $\eta(u_\ell)$ belongs to the spine, then $f(\eta(u_s))$ has to be a descendant of $\phi(u_s)$.*

Proof. The proof goes by induction on s. If $s = j$, then the claim immediately follows from our assumption that $f(\eta(u_j))$ is a descendant of $\phi(u_j)$. So assume correctness for $s - 1$, and we will show correctness for s as well.

By the induction hypothesis, $f(\eta(u_{s-1}))$ is a descendant of $\phi(u_{s-1})$. $\phi(u_s)$ is also a descendant of $\phi(u_{s-1})$, by the axis match property of ϕ. By the assumption in the claim, $\eta(u_{s-1})$ belongs to the spine, and therefore (by the spine's definition) $f(\eta(u_{s-1}))$ has to be on the same root-to-leaf path as $\phi(u_{s-1})$ and $\phi(u_s)$. $f(\eta(u_{s-1}))$ cannot be an ancestor of $\phi(u_s)$, because all the nodes between $\phi(u_{s-1})$ and $\phi(u_s)$ (if there are any) are artificial, while $f(\eta(u_{s-1}))$ is not an artificial node. Therefore, $f(\eta(u_{s-1}))$ either equals $\phi(u_s)$ or is a descendant of $\phi(u_s)$. In either case, $f(\eta(u_s))$, which is a descendant of $f(\eta(u_{s-1}))$ (Claim 7.12), is a descendant of $\phi(u_s)$. $\qquad\square$

We conclude from the above claim that there must be some $s > j$ so that $\eta(u_s)$ does not belong to the spine. Because, otherwise, $\eta(u_t)$ belongs to the spine and $f(\eta(u_t))$ is a descendant of $\phi(u_t)$; but for all nodes x in the spine, $f(x)$ is an ancestor of $\phi(w_1) = \phi(u_t)$.

So let s be the first (among the indices bigger than j) to satisfy the condition that $\eta(u_s)$ does not belong to the spine. By Claim 7.13, $f(\eta(u_s))$ has to be a descendant of $\phi(u_s)$. Since $\eta(u_s)$ does not belong to the spine, then by Observation 7.7, $E_{\eta(u_s)}$ is isomorphic to $D_{f(\eta(u_s))}$. Furthermore, $\eta(u_s)$ has a matching with u_s, and therefore by Lemma 6.2, also $f(\eta(u_s))$ has a matching with u_s. We thus obtained a node in the canonical document D (namely, $f(\eta(u_s))$) that has a matching with the node u_s but is a descendant of $\phi(u_s)$. This contradicts Proposition 6.16. $\qquad\square$

7.3 Document depth

In this section we extend the document depth lower bound (Theorem 4.6) to arbitrary redundancy-free queries that contain a query like `a/b` as a sub-query.

Remark. Some queries, like `//a`, `*/a`, or `a/*` can be evaluated with only 1 bit of memory, regardless of the document's depth. We are able to prove the document depth lower bound for queries that consist of at least one node with a child axis s.t. both this node and its parent are not wildcards. This does not cover queries that consist solely of nodes with a descendant axis, like `//a//b`. It is left open to decide whether the document depth lower bound holds for such queries as well.

Theorem 7.14. *Let Q be any redundancy-free query that has at least one node u s.t. (1) u has a child axis; (2) the node tests of u and its parent are not wildcard. Then, for any streaming algorithm that computes* BOOLEVAL$_Q$, *and for any sufficiently large integer d, there is at least one document of depth at most d, on which the algorithm requires $\Omega(\log d)$ bits of space.*

Proof. We create from BOOLEVAL$_Q$ a two-argument function BOOLEVAL$_Q^3$ (recall our notations from Section 3.2): its first argument is a 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 3.2. We thus need to create a set S of $t = \Omega(d)$ 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.
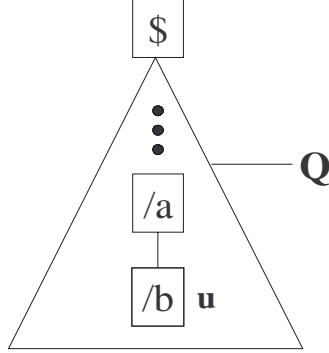

Figure 16: Schematic illustration of a query considered in the proof of Theorem 7.14

See Figure 16 for a schematic illustration of the query Q considered in this proof. Let $D = D_c$ be the canonical document corresponding to Q and let $\phi = \phi_c$ be the canonical matching of D with Q. We split the event stream representing D into three parts:

1. α is the prefix of the stream until the `startElement` event of the element $\phi(u)$.

2. β is the segment containing the element $\phi(u)$.

3. γ is the remainder of the stream.

Let Z be an "auxiliary name" (i.e., Z does not occur as a node test in Q). Let s be the depth of D. We assume that $d \geq 2s$ and define $t = d - s$ (note that $t = \Omega(d)$). For each $i = 1, \ldots, t$, we define the following three SAX event sequences:

1. $\alpha_i = \alpha \langle Z \rangle^i$.

2. $\beta_i = \langle /Z \rangle^i \beta \langle Z \rangle^i$.

3. $\gamma_i = \langle /Z \rangle^i \gamma$.

D_i is defined to be the document corresponding to the sequence $\alpha_i \circ \beta_i \circ \gamma_i$. Note that D_i is identical to D, except that we attach to it two paths of length i, all of whose nodes are named by the name "Z": one just before $\phi(u)$ and the other right after it (see Figure 17).

Example. Suppose $Q = $ /a/b. In this case u is the node named "b". D_i is the document $\langle$a$\rangle\langle$Z$\rangle^i\langle$/Z$\rangle^i\langle$b/$\rangle\langle$Z$\rangle^i\langle$/Z$\rangle^i\langle$/a$\rangle$.
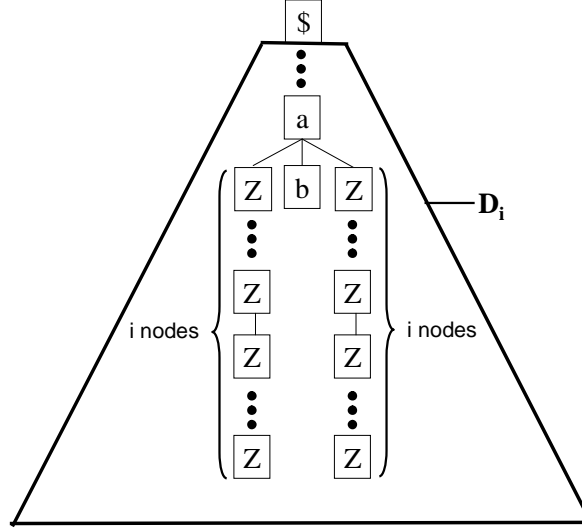

Figure 17: The document D_i

First, we claim that $D_1, \ldots, D_t$ all match Q. To this end, for each $i = 1, \ldots, t$, we need to exhibit a matching of D_i with Q. Note that the canonical document D embeds in the document D_i. In fact, it is easy to verify there is an injective homomorphism from D to D_i. Therefore, D_i matches Q (Proposition 6.17).

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 $Q = $ /a/b. For this query, $D_{i,j} = \langle$a$\rangle\langle$Z$\rangle^i\langle$/Z$\rangle^j\langle$b/$\rangle\langle$Z$\rangle^j\langle$/Z$\rangle^i\langle$/a$\rangle$. That is, $\phi(u)$ (named "b" in this example) 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. The following lemma shows that the same holds in the general case:

Lemma 7.15. *If $i > j$, then $D_{i,j}$ is a well-formed document and does not match Q.*

We conclude that S is indeed a fooling set of size $t = d - s$. Applying Theorem 3.9 to the function BOOLEVAL$_Q^3$ and Lemma 3.7 give us a space lower bound of $(\log t)/2 = \Omega(\log d)$. $\square$

Proof of Lemma 7.15. Figure 18 provides a schematic illustration of the canonical document D. For brevity, we denote the node $\phi(\text{PARENT}(u))$ by x^* and the node $\phi(u)$ by y^*. The sub-tree of D rooted at y^* is called D_1. The document obtained from D after removing the sub-tree D_1 is called D_0.
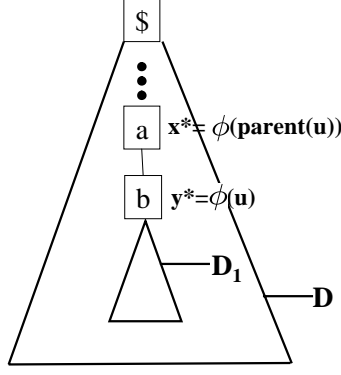
Figure 18: The document D

To simplify notation, we denote $E = D_{i,j}$. The event sequence representing E is the following:

$$\alpha \langle \mathsf{Z} \rangle^i \langle / \mathsf{Z} \rangle^j \beta \langle \mathsf{Z} \rangle^j \langle / \mathsf{Z} \rangle^i \gamma.$$

It it easy to verify that because $i > j$ the proper nesting of elements is maintained, and therefore E is well-formed.

The document E (see Figure 19) is identical to the document D, except that it has $i + j$ extra nodes named "Z". We call these nodes the "auxiliary nodes" and denote them collectively by Z. $k \stackrel{\text{def}}{=} i - j$ of these auxiliary nodes, denoted $z_1, \ldots, z_k$, separate the "copies" of x^* and y^* in E. The rest are organized as two length j paths that dangle from z_k.
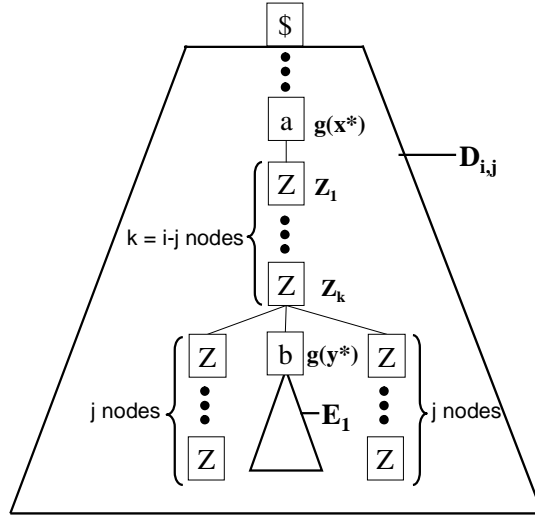

Figure 19: The document $D_{i,j} = E$

Let E_1 be the subtree of E rooted at the "copy" of y^*. Let E_0 be the document E after

removing the subtree rooted at z_1. It is easy to verify that E_0 and D_0 are isomorphic and that E_1 and D_1 are isomorphic. Let f_0, f_1 be the corresponding isomorphisms from E_0 to D_0 and from E_1 to D_1, respectively. Let g_0 and g_1 be the inverse isomorphisms. Note that z_1 is a child of $g_0(x^*)$ and $g_1(y^*)$ is a child of z_k.

Let f be the following mapping from $E \setminus Z$ to D:

$$f(x) = \begin{cases} f_0(x) & \text{if } x \in E_0 \\ f_1(x) & \text{if } x \in E_1 \end{cases}.$$

f is a 1-1 function from $E \setminus Z$ onto D. Let g be the inverse mapping. The following observations are immediate from the definitions of E and f:

Observation 7.16.

1. If a node $y \in E \setminus Z$ is a descendant of a node $x \in E \setminus Z$, then also $f(y)$ is a descendant of $f(x)$.

2. If a node $y \in E \setminus Z$ is a child of a node $x \in E \setminus Z$ and $y \neq g(y^*)$, then also $f(y)$ is a child of $f(x)$.

Let $x_1, \ldots, x_r$ be the path from the root of D to x^* (i.e., $x_1 = \text{ROOT}(D)$ and $x_r = x^*$). We call the sequence $S = (g(x_1), \ldots, g(x_r))$ the "spine" of E. The following observation is immediate from the definitions of E and f:

Observation 7.17. For every node $x \in E \setminus (S \cup Z)$, the subtrees E_x and $D_{f(x)}$ are isomorphic.

Suppose, to the contradiction, there exists a matching μ of E and Q. Define a node $x \in E \setminus Z$ to be "artificial" if $f(x)$ is an artificial node of D (note that the auxiliary nodes are not artificial). The proof of the following claim is identical to the proof of Lemma 6.12:

Claim 7.18. For all $u \in Q$, $\mu(u)$ is not an artificial node.

Note that auxiliary nodes in Z may belong to the image of μ.

Let $u_1, \ldots, u_t$ be the path from $\text{ROOT}(Q)$ to u (that is, $u_1 = \text{ROOT}(Q)$, $u_{t-1} = \text{PARENT}(u)$, and $u_t = u$). Consider the canonical matching $\phi = \phi_c$ of D with Q. Since $\phi(u_1), \ldots, \phi(u_{t-1})$ lie on the same root-to-leaf path in D, $\phi(u_1) = \text{ROOT}(D) = x_1$ and $\phi(u_{t-1}) = \phi(\text{PARENT}(u)) = x_r$, then $\phi(u_1), \ldots, \phi(u_{t-1})$ all belong to the path $x_1, \ldots, x_r$. It follows that the nodes $g(\phi(u_1)), \ldots, g(\phi(u_{t-1}))$ belong to the spine of E.

We next argue that there must be some $j \in \{1, \ldots, t\}$, so that $\mu(u_j) \neq g(\phi(u_j))$. Because, otherwise, $\mu(u) = \mu(u_t) = g(\phi(u_t)) = g(y^*)$, while $\mu(\text{PARENT}(u)) = \mu(u_{t-1}) = g(\phi(u_{t-1})) = g(x^*)$. Recall that $g(y^*)$ is a proper descendant of $g(x^*)$ (following a chain of k auxiliary nodes). Nevertheless, $\text{AXIS}(u) = \texttt{child}$, hence $\mu(u)$ must be a child of $\mu(\text{PARENT}(u))$. We

conclude that there is a j, so that $\mu(u_j) \neq g(\phi(u_j))$. Let j be the first to satisfy this. Note that $j > 1$, because $\mu(u_1) = \text{ROOT}(E) = g(\text{ROOT}(D)) = g(\phi(u_1))$. We split the analysis into four cases:

1. $\mu(u_j) \in S$.

2. $\mu(u_j) \in Z$.

3. $\mu(u_j) = g(y^*)$.

4. $\mu(u_j) \in E \setminus (S \cup Z \cup \{g(y^*)\})$.

Since Case 1 is the hardest, we postpone it to the end. Consider then Case 2. Suppose $\mu(u_j) = z$ for some auxiliary node z. Since z is named by an auxiliary name, u_j must have a wildcard node test. This already implies that $j < t$, because $u_t = u$ does not have a wildcard node test. By the choice of j, $\mu(u_{j-1}) = g(\phi(u_{j-1}))$. Since $j < t$, $\phi(u_{j-1})$ is one of the nodes $x_1, \ldots, x_{r-1}$ (recall that $\phi(u_{t-1}) = x_r$). z_1 is a child of $g(x_r)$ and all the auxiliary nodes are descendants of z_1. We this obtain that z must be a proper descendant of $\mu(u_{j-1})$, and thus u_j has a descendant axis. This is impossible, because Q is star-restricted.

Consider Case 3. We have $\mu(u_j) = g(y^*)$. Since $g(y^*)$ is a proper descendant of each of the nodes in the spine and since $\mu(u_{j-1}) = g(\phi(u_{j-1}))$ belongs to the spine, then u_j has a descendant axis. This implies that $j < t$, because $u_t = u$ has a child axis. The restriction of μ to the subtree $E_{g(y^*)}$ gives a matching of $g(y^*)$ with u_j. Since $g(y^*)$ does not belong to the spine, the subtrees $E_{g(y^*)}$ and D_{y^*} are isomorphic (Observation 7.17). Therefore, there is also a matching of y^* with u_j (Lemma 6.2). Since $j < t$, y^* is a descendant of $\phi(u_j)$. We thus found a node in Q (namely, u_j) whose shadow (namely, $\phi(u_j)$) has a descendant (namely, y^*) that has a matching with u_j. This contradicts Proposition 6.16.

Consider now Case 4. By Observation 7.17, since $\mu(u_j)$ does not belong to the spine, $E_{\mu(u_j)}$ and $D_{f(\mu(u_j))}$ are isomorphic. The restriction of μ to $E_{\mu(u_j)}$ gives a matching of $\mu(u_j)$ with u_j. Hence, there also exists a matching η of $f(\mu(u_j))$ with u_j (Lemma 6.2).

We would like to use η and the canonical matching ϕ to construct a hybrid matching ϕ' of Q and D. To this end, we need to make sure that $f(\mu(u_j))$ relates to $\phi(u_{j-1})$ according to $\text{AXIS}(u_j)$. By the choice of j, $f(\mu(u_{j-1})) = \phi(u_{j-1})$. If u_j has a child axis, then $\mu(u_j)$ is a child of $\mu(u_{j-1})$. Since $\mu(u_j), \mu(u_{j-1}) \notin Z$ and $\mu(u_j) \neq g(y^*)$ then by Observation 7.16, also $f(\mu(u_j))$ is a child of $f(\mu(u_{j-1})) = \phi(u_{j-1})$. If u_j has a descendant axis, then since $\mu(u_j), \mu(u_{j-1}) \notin Z$, by Observation 7.16 also $f(\mu(u_j))$ is a descendant of $f(\mu(u_{j-1})) = \phi(u_{j-1})$. We conclude that $f(\mu(u_j))$ indeed relates to $\phi(u_{j-1})$ according to $\text{AXIS}(u_j)$.

We now apply Lemma 6.7, and obtain that the hybrid mapping ϕ' induced by ϕ and η is a matching of Q and D. Note that ϕ' is a non-canonical matching, because $\phi'(u_j) = $

56

$f(\mu(u_j)) \neq \phi(u_j)$. This contradicts Lemma 6.15, according to which the canonical matching is the only matching of Q and D.

Finally, let us go back to Case 1. Recall that $g(\phi(u_1)), \ldots, g(\phi(u_{t-1}))$ belong to the spine. Since $\mu(u_j) \neq g(\phi(u_j))$ but $\mu(u_j)$ belongs to the spine, then $\mu(u_j)$ is either an ancestor or a descendant of $g(\phi(u_j))$. An identical argument to the one done in the proof of Claim 7.9 shows that $\mu(u_j)$ cannot be an ancestor of $g(\phi(u_j))$. So assume it is a descendant. We will need the following claim:

Claim 7.19. *Let $j \leq s \leq t$. If for all $j \leq \ell < s$, $\mu(u_\ell)$ belongs to the spine, then $f(\mu(u_s))$ has to be a descendant of $\phi(u_s)$.*

The proof is identical to the proof of Claim 7.13 and thus is not repeated.

If for all $s > j$, $\mu(u_s)$ belongs to the spine, then in particular $\mu(u_t)$ does. It follows from the claim that $f(\mu(u_t))$ is a descendant of $\phi(u_t)$. Therefore, also $\mu(u_t)$ is a descendant of $g(\phi(u_t))$. However, $g(\phi(u_t)) = g(\phi(u)) = g(y^*)$ is a descendant (not an ancestor) of all the nodes in the spine.

So there must be some $s > j$ so that $\mu(u_s)$ does not belong to the spine. Assume that s is the first to satisfy this condition. By Claim 7.19, $f(\mu(u_s))$ has to be a descendant of $\phi(u_s)$. We know that $\mu(u_s)$ does not belong to the spine. If it does not belong to Z as well, then by Observation 7.17 $E_{\mu(u_s)}$ is isomorphic to $D_{f(\mu(u_s))}$. The restriction of μ to $E_{\mu(u_s)}$ gives a matching of $\mu(u_s)$ with u_s. Therefore, there is also matching of $f(\mu(u_s))$ with u_s (Lemma 6.2). We thus obtained a node in Q (namely, u_s) whose shadow in D (namely, $\phi(u_s)$) has a descendant (namely, $f(\mu(u_s))$) that has a matching with u_j. This contradicts Proposition 6.16.

We are left to address the case $\mu(u_s) \in Z$. Suppose $\mu(u_s) = z$ for some auxiliary node z. Since z has an auxiliary name as a name, u_s must have a wildcard node test. It follows that $s < t - 1$ (because both $u_t = u$ and $u_{t-1} = \text{PARENT}(u)$ do not have a wildcard node test). Moreover, since Q is star-restricted, all the children of u_s, and in particular u_{s+1}, have a child axis.

If z is one of the auxiliary nodes on the two length j paths that dangle from z_k, then all its descendants are auxiliary nodes too, and hence none of them can match the nodes u_{t-1} and u_t, which do not have a wildcard node test. This means that $\mu(u_{t-1})$ and $\mu(u_t)$ are not descendants of $\mu(u_s)$, contradicting the axis match property of μ.

So assume $z = z_\ell$ for some $\ell \in \{1, \ldots, k\}$. If $\ell < k$, then the only child of z_ℓ is $z_{\ell+1}$, and therefore $\mu(u_{s+1}) = z_{\ell+1}$. Continuing inductively, we obtain that for each $p = 1, \ldots, k - \ell$, $\mu(u_{s+p}) = z_{\ell+p}$, and u_{s+p} has a wildcard node test and a child axis. Note that $s + k - \ell$ has to be smaller than $t - 1$, because u_{t-1} does not have a wildcard node test. Now, since $\mu(u_{s+k-\ell}) = z_k$, since $\mu(u_{t-1})$ is a non-auxiliary descendant of $\mu(u_{s+k-\ell})$, and since all the

non-auxiliary descendants of z_k belong to the subtree $E_{g(y^*)}$, $\mu(u_{t-1})$ belongs to this subtree. This means that $\mu(u_{t-1})$ does not belong to the spine, and thus $E_{\mu(u_{t-1})}$ and $D_{f(\mu(u_{t-1}))}$ are isomorphic (Observation 7.17). $\mu(u_{t-1})$ has a matching with u_{t-1}, and so also $f(\mu(u_{t-1}))$ has a matching with u_{t-1} (Lemma 6.2). Since $\mu(u_{t-1})$ belongs to the subtree rooted at $g(y^*)$, it is a descendant of $g(x^*)$. It follows that also $f(\mu(u_{t-1}))$ is a descendant of x^*. We thus obtained a node in Q (namely, u_{t-1}) whose shadow (namely, x^*) has a descendant (namely, $f(\mu(u_{t-1}))$) that has a matching with u_{t-1}. This contradicts Proposition 6.16. $\qquad\square$

8 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 leaf-only-value-restricted univariate conjunctive query. An example run is provided in the end of the section.

Overview   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 a matching ϕ of D with Q. It declares D as a match to Q if and only if the construction ends successfully. The algorithm has the property that if there is at least one matching of D with Q, then the algorithm can always find it, while if there are no matchings of D with Q, the construction fails. It then follows from Lemma 5.10 that the algorithm's output is always correct.

The algorithm is event-driven. As SAX events arrive, corresponding event handlers are called, updating the global variables of the algorithm. There are five event handlers: `startDocument()`, `endDocument()`, `startElement(n)`, `endElement(n)`, and `text(`α`)`. In addition, the algorithm has several other subroutines, described below.

The algorithm gradually constructs the matching ϕ 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" (see definition below). 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" (see definition below) for u or not. The algorithm declares the document as matching the query if and only if a real match for the query root is found. The mapping of query nodes to their real matches is the desired matching ϕ.

More formally, a document node x is a *candidate match for u*, if: (1) NAME(x) passes NTEST(u); and (2) x relates to the candidate match of PARENT(u) according to AXIS(u).[3] x

---

[3]This definition holds for nodes $u \neq$ ROOT(Q). x is a candidate match for ROOT(Q), only if $x =$ ROOT(D).

is also a *real match for* u, if either (1) u is a leaf and STRVAL(x) belongs to TRUTH(u);[4] or (2) u is an internal node and every child v of u has a real match y that relates to x according to AXIS(v). It is easy to verify that if x is a real match for u, then the map ϕ_u that maps u and its descendants to their corresponding real matches in the subtree D_x is a matching of x with u. In particular, if the document root is determined as a real match for the query root, the corresponding map is matching of D with Q.

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 PARENT(u), we know whether x relates to z according to 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 v. This can be inferred only at the `endElement` event of x.

Global variables   The algorithm maintains the following global variables:

1. `frontier`: A table consisting of the current query "frontier". Each tuple in the table has the following attributes:

   (a) `ref`: A reference to a query node.

   (b) `matched`: A flag indicating whether a real match has already been found for the query node.

   (c) `level`: The document level at which we expect to find a candidate match for the query node.

   (d) `strValueStart`: The position in the buffer (see below), in which the string value of the candidate match for the query node begins.

2. `buffer`: A buffer consisting of text from the document. Has the following data members:

   (a) `data`: The content of the buffered text.

   (b) `size`: The size of the buffer.

   (c) `refCount`: The number of currently processed document nodes whose string value is being buffered.

3. `currentLevel`: The level of the currently processed document node.

---

[4]Recall that Q is leaf-only-value-restricted, and thus only leaves need to satisfy the value match property.

Event handlers `startDocument` (see Figure 20) initializes the global variables. In particular, it inserts the query root to the frontier (lines 2-3), setting its `matched` flag to `false`, indicating that no real match for the root has been found yet, and its `level` to 0, indicating that a candidate match for the root should be at level 0. `currentLevel` is initialized to indicate that the current document level is 0 (line 7).

```
Function startDocument()
1:  frontier.initialize()
2:  insert a new record with the following values into frontier:
3:    (ref := ROOT(Q), matched := false, level := 0)
4:  buffer.data := ""
5:  buffer.size := 0
6:  buffer.refCount := 0
7:  currentLevel := 0


Function startElement(n)
1:  select * from frontier where
2:    (ref.ntest = n OR ref.ntest = *) AND
3:    (ref.axis = descendant OR level = currentLevel) AND
4:    (matched = false)
5:  for all records u selected do
6:    if (u.ref.isLeaf) then
7:      buffer.refCount := buffer.refCount + 1
8:      u.strValueStart := buffer.size
9:    else
10:     if (u.ref.axis = child)
11:       delete u from frontier
12:     for v in u.ref.children do
13:       insert a new record with the following values into frontier:
14:         (ref := v, matched := false, level := currentLevel + 1)
15:     end for
16:   end if
17: end for
18: currentLevel := currentLevel + 1


Function text(α)
1:  if (buffer.refCount > 0)
2:    append α to buffer
```

Figure 20: Pseudo-code for the functions `startDocument`, `startElement`, and `text`

$\quad$`startElement` (see Figure 20) is called every time a new document node x starts. The function first selects all the query nodes u in the frontier, for which x is a candidate match, and which have not found a real match yet (lines 1-4). Recall that x is a candidate match for u if: (1) NAME(x) passes NTEST(u) (line 2); and (2) x relates to the candidate match of PARENT(u) according to AXIS(u). We next explain how line 3 guarantees the latter condition. Let z be the candidate match for PARENT(u). As described below, u is inserted into the frontier at the `startElement` event of z and removed at the `endElement` event of z. Therefore, since u is currently in the frontier, x must be "encapsulated" by z, i.e., x is a descendant of z. If AXIS(u) = `descendant`, this already suffices to guarantee Condition (2). If AXIS(u) = `child`, x must be a child of z. When u is inserted into the frontier, its

`level` attribute is set to be one more than the level of z. Thus, if the current level equals the `level` attribute of u, it must mean that x is a child of z.

If a real match for a node u has already been found (line 4), we do not need any additional real matches for u, and thus we do not need to verify whether x turns into a real match or not.

If u is a leaf, checking whether x turns into a real match or not will require inspecting the string value of x. We thus start buffering the contents of the `text` node descendants of x (lines 6-8).

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 D_x. We thus insert all the children of u into the frontier (lines 12-15), setting their `matched` flag to `false` and their `level` to the next document level.

When u has a `child` axis, we know that no further candidate matches for u can be found among descendants of x. We can thus temporarily remove u from the frontier, to save space (lines 10-11). u will be put back into the frontier, when the element x ends. Note that this optimization cannot be applied if u has a `descendant` axis, because if the document is recursive, then both x and descendants of x can be candidate matches for u at the same time.

The function `text` (see Figure 20) updates the buffer with the text content of the current `text` node, if there are any "consumers" for this buffer.

`endElement` (see Figure 21) is called every time a document node x ends. After updating the current document level (line 1), the function selects all the leaf nodes in the frontier, for which x is a candidate match and which have not found a real match yet (lines 2-6). For each such node u, x is a real match for u if and only if $\text{STRVAL}(x) \in \text{TRUTH}(u)$. In lines 7-10, the function extracts the string value of x from the buffer and then checks whether it belongs to $\text{TRUTH}(u)$. We use the function $decrementRefCount()$ to decrement the reference count of the buffer and reset the *data* and *size* members when the reference count reaches 0. The membership in $\text{TRUTH}(u)$ is done by invoking the function `evalPredicate`, whose pseudo-code is omitted. Let w be the succession root of u. There are two cases: either $w = \text{ROOT}(Q)$ or w is a predicate child of $\text{PARENT}(w)$, and is thus a variable in an atomic predicate P of $\text{PREDICATE}(\text{PARENT}(w))$. If $w = \text{ROOT}(Q)$, then $\text{TRUTH}(u) = \mathcal{S}$, and thus `evalPredicate` returns the value `true`. If w is a variable in P, then $\text{TRUTH}(u) = \text{TRUTH}(P)$ (recall Definition 5.6). Thus whether $\text{STRVAL}(x) \in \text{TRUTH}(u)$ depends on whether the evaluation of the atomic predicate P on the value $\text{STRVAL}(x)$ results in the value `true`. This evaluation is done by the function `evalPredicate`.

Next, the function `endElement` addresses the internal nodes u, for which x is a candidate

```
Function endElement(n)
1:  currentLevel := currentLevel - 1
2:  select * from frontier where
3:    (ref.ntest = n OR ref.ntest = *) AND
4:    (ref.axis = descendant OR level = currentLevel) AND
5:    matched = false AND
6:    ref.isLeaf
7:  for all records u selected do
8:    u.matched := evalPredicate(u, buffer.data[u.strValueStart,buffer.size])
9:    buffer.decrementRefCount()
10: end for
11: select * from frontier where
12:   level > currentLevel
13:   group by ref.parent
14: for all ref.parent u of records selected do
15:   m := true
16:   for all records v that were selected and for which ref.parent = u do
17:     if (v.matched = false)
18:       m := false
19:     delete v from frontier
20:   end for
21:   if (u.axis = descendant) then
22:     get record urec from frontier where ref = u
23:   else
24:     create a new record urec with:
25:       (ref := u, matched := false, level := currentLevel)
26:     insert urec into frontier
27:   end if
28:   urec.matched := m
29: end for

Function endDocument()
1:  get record r from frontier where ref = ROOT(Q)
2:  return r.matched
```

Figure 21: Pseudo-code for the functions `endElement` and `endDocument`

match. In lines 11-13, the function selects all the children of these nodes. x is a real match for u if and only if all the children v of u have already found a real match (lines 15-20). These children are then removed from the frontier (line 19).

u itself may or may not be at the frontier at this point. If u has a `descendant` axis, it already exists in the frontier (lines 21-22). If u has a `child` axis, it should be reinserted into the frontier (lines 23-27). The `matched` flag of u is set in accordance with whether x is a real match for u or not (line 28).

Finally, the function `endDocument` (see Figure 21) returns the output of the algorithm, which is the value of the `matched` flag of the query root.

Example run In Figure 22 we present an example of how the algorithm processes the query Q = `/a[c[.//e and f] and b]`. We show a sample document and a snapshot of the state of the main global variables after each event. We use tuples to represent the values of the `level`,`ref.ntest`, and `matched` attributes of the records in the frontier table. Each

event is represented by its name and by the level it happened. The first interesting event is `startElement(d)` (event 4). Since the "d" element does not match any node in the frontier, we increase the level by one but keep the frontier intact. The other interesting event is the second `startElement(c)` event (event 11). Since the "c" query node is already matched at that point, instead of processing the new "c" document node, we ignore it and simply increment the level variable. At the end of the processing, the `matched` flag of the root is set to 1 (`true`), meaning that the document matched the query.
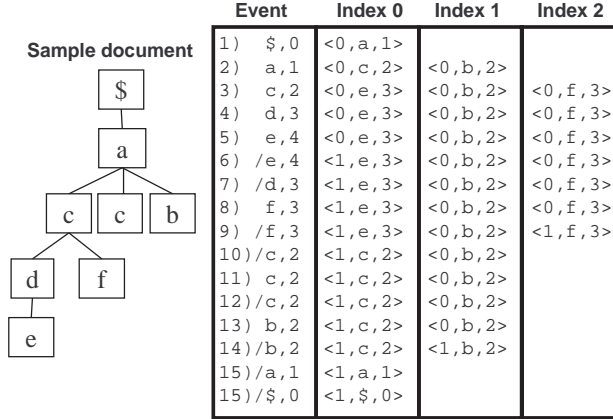
	Event	Index 0	Index 1	Index 2
Sample document	1) $,0	<0,a,1>		
	2) a,1	<0,c,2>	<0,b,2>	
	3) c,2	<0,e,3>	<0,b,2>	<0,f,3>
	4) d,3	<0,e,3>	<0,b,2>	<0,f,3>
	5) e,4	<0,e,3>	<0,b,2>	<0,f,3>
	6) /e,4	<1,e,3>	<0,b,2>	<0,f,3>
	7) /d,3	<1,e,3>	<0,b,2>	<0,f,3>
	8) f,3	<1,e,3>	<0,b,2>	<0,f,3>
	9) /f,3	<1,e,3>	<0,b,2>	<1,f,3>
	10)/c,2	<1,c,2>	<0,b,2>	
	11) c,2	<1,c,2>	<0,b,2>	
	12)/c,2	<1,c,2>	<0,b,2>	
	13) b,2	<1,c,2>	<0,b,2>	
	14)/b,2	<1,c,2>	<1,b,2>	
	15)/a,1	<1,a,1>		
	15)/$,0	<1,$,0>		

Sample document tree: $ → a → { c, c, b }; first c → { d, f }; d → e.

Figure 22: Example run for `/a[c[.//e and f] and b]`

Correctness   The correctness of the algorithm follows from the next theorem and from Lemma 5.10. The proof is straightforward from the description above and from the definition of matchings, and is thus omitted.

Theorem 8.1 (Correctness). *Let Q be a leaf-only-value-restricted univariate conjunctive query and let D be a document. If D has at least one matching with Q, then the algorithm, when running on D and Q, finds a real match for $\text{ROOT}(Q)$. Conversely, if the algorithm finds a real match for $\text{ROOT}(Q)$, then there exists a matching of D with Q.*

Complexity   In order to analyze the time and space complexities of the algorithm, we need to develop some terminology. We first introduce the following weak notion of a matching:

Definition 8.2 (Path matching). Let Q be a query and let D be a document. A *path matching* of a node $x \in D$ with a node $u \in Q$ is a map ρ from $\text{PATH}(u)$ to $\text{PATH}(x)$, which satisfies the following:

1. **Root match:** $\rho(\text{ROOT}(Q)) = \text{ROOT}(D)$.

2. **Axis match:** For all nodes $v \in \text{PATH}(u)$, $v \neq \text{ROOT}(Q)$, $\rho(v)$ relates to $\rho(\text{PARENT}(v))$ according to $\text{AXIS}(v)$.

3. **Node test match:** For all nodes $v \in \text{PATH}(u)$, $v \neq \text{ROOT}(Q)$, $\text{NAME}(\rho(v))$ passes $\text{NTEST}(v)$.

x is said to *path match* u, if there exists a path matching of x with u.

Clearly, if x matches or even structurally matches u (relative to the context $\text{ROOT}(Q) = \text{ROOT}(D)$), then it also path matches u. Note that in our algorithm, if a node x is determined as a candidate match for a node u, then x must path match u. We next redefine the notion of recursion w.r.t. path matchings.

Definition 8.3 (Path recursion depth). Let Q be a query. The *path recursion depth* of a document D w.r.t. Q is the maximum length of a sequence of nodes $x_1, \ldots, x_r$, all of which path match the same node in Q and are nested within each other (i.e., x_i is a descendant of x_{i-1}, for $i = 2, \ldots, r$).

For example, if $Q = $ `//a[b]` and $D = \langle a \rangle \langle a \rangle \langle /a \rangle \langle /a \rangle$, then the path recursion depth of D w.r.t. Q is 2, because both of the "a" nodes in D path match the "a" node in Q. Its recursion depth, however, is 0, because neither of the two "a" nodes in D matches the "a" node in Q. Clearly, the recursion depth of a document is always a lower bound on the path recursion depth of the document.

Next, we define the following new notion:

Definition 8.4 (Text width). Let Q be a leaf-only-value-restricted univariate conjunctive query. The *text width* of a document D w.r.t. Q is the maximum length, over all leaf nodes $u \in Q$, and over all nodes $x \in D$ that path match u, of $\text{STRVAL}(x)$.

For example, if $Q = $ `/a[b]` and $D = \langle a \rangle$ `dear` $\langle b \rangle$`sir`$\langle /b \rangle$ `or` $\langle b \rangle$`madam`$\langle /b \rangle$ $\langle /a \rangle$, then the text width of D w.r.t. Q is 5, because the second "b" node in D is the node with the maximum length string value that path matches a leaf in Q.

Finally, we define a strong notion of redundancy-freeness w.r.t. path matchings.

Definition 8.5 (Path consistency). Two nodes $u, v \in Q$ are said to be *path consistent*, if there exists a document D and a node $x \in D$, so that x path matches both u and v.

For example, in the query $Q = $ `/a[.//b/c and b//c]`, the two "c" nodes are path consistent: the node named "c" in the document $D = \langle a \rangle \langle b \rangle \langle c \rangle \langle /c \rangle \langle /b \rangle \langle /b \rangle$ path matches both.

Definition 8.6 (Path consistency-free query). A query Q is called *path consistency-free*, if no two of its nodes are path consistent.

Consistency-freeness is a stronger notion than subsumption-freeness. It is easy to verify that every path consistency-free query is also subsumption-free.

Definition 8.7 (Closure-free queries). A query Q is called *closure-free*, if none of its nodes has the `descendant` axis.

We are now ready to state the theorem about the space and time complexities of the algorithm. The following theorem shows that the algorithm uses space, which is: (1) (quasi-) linear in the size of the query; (2) linear in the path recursion depth; (3) logarithmic in the document depth; and (4) linear in the text width. Thus, the algorithm matches the document depth lower bound (Theorem 7.14) and almost matches the recursion depth lower bound (Theorem 7.4), modulo the difference between recursion depth and path recursion depth. The second part of the theorem shows that for queries that are path consistency-free and closure-free, the space used is (quasi-) linear in the query frontier size, and thus for these queries the algorithm matches our main lower bound (Theorem 7.1).

Theorem 8.8 (Complexity). *Let Q be any leaf-only-value-restricted univariate conjunctive query. Let D be any document of depth at most d, of path recursion depth at most r w.r.t. Q, and of text width space at most w w.r.t. Q. Then, the algorithm, when executed on Q and D, uses at most $O(|Q| \cdot r \cdot (\log|Q| + \log d + \log w) + w)$ bits of space and runs in $\tilde{O}(|D| \cdot |Q| \cdot r)$ time. ($\tilde{O}$ suppresses poly-logarithmic factors.)*

If Q is in addition path consistency-free and closure-free, then the algorithm uses at most $O(FS(Q) \cdot (\log|Q| + \log d + \log w) + w)$ bits of space and runs in $\tilde{O}(|D| \cdot FS(Q))$ time.

Proof. We start with the space complexity analysis. We need to analyze the space needed for each of the global variables used by the algorithm (the local variables require negligible space).

First, `currentLevel` needs $O(\log d)$ bits of space. Next, the `buffer` variable holds string values of document nodes that are candidate matches for query nodes. Note that even though multiple string values may be buffered simultaneously, they are always nested within each other. The size of the buffer thus never exceeds the length of the string value of the outermost node. This node is a candidate match for some query leaf node, and thus also path matches it. This string value must be then of length at most w. Thus, the buffer size is $O(w)$.

Finally, we analyze the maximum size of the frontier. A query node u is inserted into the frontier every time a candidate match is found for its parent. The frontier consists

of multiple copies of u simultaneously only if these candidate matches are nested within each other. This means that if the frontier consists of k copies of u, there are k candidate matches for PARENT(u) that are nested within each other. These candidate matches path match PARENT(u). Hence, k must be at most r, due to the fact D has path recursion depth of at most r w.r.t. Q. We conclude that each query node can have at most r copies in the frontier simultaneously. Therefore, the size of the frontier is at most $|Q| \cdot r$. Every tuple in the frontier is of size $O(\log |Q| + \log d + \log w)$, giving the stated space upper bound.

Suppose now that Q is path consistency-free and closure-free. We next argue that whenever a node u is inserted into the frontier, the frontier consists solely of nodes that are siblings of u or of one of its ancestors.

Suppose, to reach a contradiction, that when u is inserted into the frontier, there exists a node v in the frontier, which is neither a sibling of u nor a sibling of an ancestor of u. Let y be the document node, during whose `startElement` event v was inserted into the frontier. This means that y is a candidate match for PARENT(v). Similarly, let x be the document node, during whose `startElement` event u is inserted into the frontier. Since v is still in the frontier at the `startElement` event of x, x must be a descendant of y.

y is the candidate match for PARENT(v) and thus path matches PARENT(v). x is the candidate match for PARENT(u) and thus path matches PARENT(u). Since x is a descendant of y and no ancestor of PARENT(u) has a `descendant` axis (recall that Q is closure-free), then there must be an ancestor w of PARENT(u) that path matches y. Since Q is path consistency-free, $w = $ PARENT(v), because otherwise y path matches two distinct nodes in Q.

We obtained that PARENT(v) is an ancestor of u. Thus, the only way for v not to be a sibling of u or of one of its ancestors, is that v itself is an ancestor of u. Let z be the child of y succeeding y on the path segment PATH($y..x$). Since v is an ancestor of u and has a `child` axis, z must path match v. Therefore, at the `startElement` event of z, v should have been removed from the frontier (lines 10-11). That did not happen, hence v cannot be an ancestor of u.

We conclude that the frontier of the algorithm always consists of the "query frontier" (in the sense of Definition 4.1) of the query at the last node to be inserted into the frontier. Therefore, its size never exceeds FS(Q), implying the stated space upper bound.

The running time of the algorithm is dominated by the time it takes to process the `startElement` and `endElement` events. Each one of these runs a loop on nodes selected from the frontier. Since the size of the frontier is at most $|Q| \cdot r$ for general queries and FS(Q) for path consistency-free closure-free queries, the stated time upper bounds follow. $\square$

9   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 lower bounds.

This work should be viewed only as a starting point for the study of the memory requirements of XPath evaluation on streams. First, our bounds are tight only for a restricted class of queries. Second, there are other sources for high memory use, which we have not addressed at all, such as full-fledged evaluation of queries with predicates (as opposed to just filtering such queries) and evaluation of XQuery queries with multiple output nodes. In subsequent work [5], we prove bounds for the former. The latter remains an open problem.

Acknowledgments

References

[1] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases (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 *Proceedings of the 21st ACM Symposium on Principles of Database Systems (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 *Proceedings of the 1st Workshop on Programming Languages for XML (PLAN-X)*, 2002.

[4] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. In *Proceedings of the 23rd ACM Symposium on Principles of Database Systems (PODS)*, pages 177–188, 2004.

[5] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in query evaluation over XML streams. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems (PODS)*, pages 216–227, 2005.

[6] 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.

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

[8] 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.

[9] D. Chamberlin, P. Fankhauser, D. Florescu, M. Marchiori, and J. Robie. *XML Query Use Casese.* W3C, `http://www.w3.org/TR/XML/query`, 2005.

[10] C. Y. Chan, P. Felber, M. N. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 235–244, 2002.

[11] B. Choi, M. Mahoui, and D. Wood. On the optimality of holistic algorithms for twig queries. In *Proceedings of the 14th International Conference on Database and Expert Systems Applications (DEXA)*, pages 28–37, 2003.

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

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

[14] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, pages 341–342, 2002.

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

[16] 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.

[17] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proceedings of the 22nd ACM Symposium on Principles of Database Systems (PODS)*, pages 179–190, 2003.

[18] T. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory (ICDT)*, pages 173–189, 2003.

[19] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, pages 1076–1088, 2005.

[20] A. K. Gupta and D. Suciu. Stream processing of XPath queries with predicates. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 419–430, 2003.

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

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

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

[24] 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.

[25] D. Olteanu, T. Kiesling, and F. Bry. An evaluation of regular path expressions with qualifiers against XML streams. In *Proceedings of the 19th International Conference on Data Engineering (ICDE)*, pages 702–704, 2003.

[26] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the 22nd ACM SIGMOD International Conference on Management of Data*, pages 431–442, 2003.

[27] L. Segoufin. Typing and querying XML documents: Some complexity bounds. In *Proceedings of the 22nd ACM Symposium on Principles of Database Systems (PODS)*, pages 167–178, 2003.

[28] M. Y. Vardi. The complexity of relational query languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.

[29] A. C-C. Yao. Some complexity questions related to distributive computing. In *Proceedings of the 11th ACM Symposium on Theory of Computing (STOC)*, pages 209–213, 1979.

A   Redundancy-free XPath

In this section we prove statements about the various query classes discussed in Section 5.

Lemma 5.10 (restated) *A document D matches a query Q if and only if there exists a matching of D and Q.*

In order to prove the lemma, we will need the two following lemmas:

Lemma A.1. *Let Q be a univariate query and let D be a document. If there exists a matching ϕ of a $u \in Q$ with a $x \in D$, then u, x, and ϕ must satisfy the three following conditions:*

1. *If $u \neq \text{ROOT}(Q)$, $\text{NAME}(x)$ passes $\text{NTEST}(u)$, and if $u = \text{ROOT}(Q)$, $x = \text{ROOT}(D)$.*

2. *x satisfies $\text{PREDICATE}(u)$.*

3. *For every node $v \in Q_u$, $\phi(v) \in \text{SELECT}(v|u = \phi(u))$.*

Lemma A.2. *Let Q be a leaf-only-value-restricted univariate query and let D be a document that matches Q. For every nodes $u \in Q$, $x \in D$, and $y \in D$, which satisfy the following three conditions:*

1. *if $u \neq \text{ROOT}(Q)$, $\text{NAME}(x)$ passes $\text{NTEST}(u)$, and if $u = \text{ROOT}(Q)$, $x = \text{ROOT}(D)$;*

2. *x satisfies $\text{PREDICATE}(u)$;*

3. *y belongs to $\text{SELECT}(\text{LEAF}(u)|u = x)$ and $\text{STRVAL}(y) \in \text{TRUTH}(\text{LEAF}(u))$;*

there exists a matching ϕ of x with u, which satisfies $\phi(\text{LEAF}(u)) = y$.

Proof of Lemma 5.10. Suppose D matches Q. Then, by Definition 3.6, $\text{ROOT}(D)$ satisfies the predicate of $\text{ROOT}(Q)$ and $\text{SELECT}(\text{LEAF}(\text{ROOT}(Q))|\text{ROOT}(Q) = \text{ROOT}(D)) \neq \emptyset$. Let y be some node in this selection. Since $\text{TRUTH}(\text{LEAF}(\text{ROOT}(Q))) = \mathcal{S}$, then $\text{STRVAL}(y) \in \text{TRUTH}(\text{LEAF}(\text{ROOT}(Q)))$ trivially. It now follows from Lemma A.2 that there exists a matching ϕ of $\text{ROOT}(D)$ with $\text{ROOT}(Q)$. ϕ is the desired matching of D with Q.

For the other direction, assume there exists a matching ϕ of $\text{ROOT}(D)$ with $\text{ROOT}(Q)$. Then it follows from Lemma A.1 that: (1) $\text{ROOT}(D)$ satisfies $\text{PREDICATE}(\text{ROOT}(Q))$; and (2) $\phi(\text{LEAF}(\text{ROOT}(Q))) \in \text{SELECT}(\text{LEAF}(\text{ROOT}(Q))|\text{ROOT}(Q) = \text{ROOT}(D))$. In particular, $\text{SELECT}(\text{LEAF}(\text{ROOT}(Q))|\text{ROOT}(Q) = \text{ROOT}(D)) \neq \emptyset$, and thus D matches Q. □

The following is a basic fact describing necessary and sufficient conditions for an atomic predicate to be satisfied. It will play a crucial role in the proofs of the two above lemmas.

Proposition A.3. *Let $u \in Q$ be any node, let P be any of the constituent atomic predicates in* PREDICATE(u), *let r_P be the root of P, and let w be the child of u that occurs as a single variable in P. Then, for every node $x \in D$, **EBV**(PEVAL$(r_P, x)) = $* true *if and only if there exists a node $y \in$ SELECT(LEAF$(w)|u = x)$ whose string value belongs to* TRUTH(LEAF(w)).

Proof. Let s be the leaf of P, which is labeled by a pointer to w. Recall from Definition 5.6 that TRUTH(LEAF(w)) = TRUTH(P).

Let $S = $ PEVAL(s, x). Recall from Definition 3.5 that S is the sequence of string values of the nodes in SELECT(LEAF$(w)|u = x)$. The statement in the proposition boils down to proving that $S \cap$ TRUTH$(P) \neq \emptyset$ (where we view here S as a set).

Let s' be the last ancestor of s (climbing from s upwards), whose output is non-boolean. By our definition of the predicate evaluation process (Definition 3.5), PEVAL(s', x) is the sequence of values we would obtain by substituting the values in S in the expression corresponding to s'. Let us denote this sequence by S'.

According to the definition of atomic predicates (Definition 5.3), we have two possible cases: either (1) $s' = r_P$, implying the root itself has a non-boolean output; or (2) r_P has a boolean output and non-boolean arguments and s' is a child of r_P.

In the first case, **EBV**(PEVAL$(r_P, x)) = $ true if and only if $S' \neq \emptyset$, which in turns happens if and only if $S \neq \emptyset$. Note that in this case TRUTH$(P) = \mathcal{S}$, and thus the condition $S \neq \emptyset$ is equivalent to the condition $S \cap$ TRUTH$(P) \neq \emptyset$.

Consider then the second case. In this case r_P is a boolean operator with non-boolean operands. All the children of r_P, except for s', do not depend on w and thus evaluate to constant atomic values. Therefore, by Definition 3.5, PEVAL$(r_P, x) = $ true if and only if there exists a value in S', which makes P true. But this in turn happens iff one of the values in S belongs to TRUTH(P). Thus, PEVAL$(r_P, x) = $ true iff $S \cap$ TRUTH$(P) \neq \emptyset$, as desired. $\qquad\square$

Proof of Lemma A.1. The first condition is trivially satisfied by the root match and node test match properties of ϕ. We prove the satisfaction of the two other conditions by a double induction on the following quantities:

1. HEIGHT(u), which is 0 if u is a leaf, and $\max_w$ HEIGHT$(w) + 1$, where the maximum is over the children w of u, if u is an internal node.

2. DIST(u, v), which is 0 if $v = u$, and DIST$(u,$ PARENT$(v)) + 1$ otherwise.

To begin the induction, suppose $\text{HEIGHT}(u) = 0$. That is, u is a leaf, and thus in particular has no children and its predicate is empty. It follows that $\phi(u)$ trivially satisfies $\text{PREDICATE}(u)$, and thus the second condition is met. v must equal u, and therefore, $\text{SELECT}(v|u = \phi(u)) = \{\phi(u)\}$. Indeed, $y = \phi(v) = \phi(u) = x$ belongs in this case to $\text{SELECT}(v|u = x)$, as desired. Therefore, also the third condition is met, and the statement holds for the induction base.

Assume, then, that the statement holds for nodes u of maximum height at most k, and we will show it holds also for nodes u of maximum height $k + 1$. We prove the following:

Claim A.4. *For all nodes $w \in Q_u$, $\phi(w)$ satisfies $\text{PREDICATE}(w)$.*

Proof. Let $P_1, \ldots, P_\ell$ be the constituent atomic predicates of $\text{PREDICATE}(w)$, let $r_{P_1}, \ldots, r_{P_\ell}$ be their roots, and let $w_1, \ldots, w_\ell$ be the children of w that appear as single variables in them, respectively. For each $i = 1, \ldots, \ell$, the following hold: (1) the restriction of ϕ to Q_{w_i} is a matching of $\phi(w_i)$ with w_i; (2) $\text{HEIGHT}(w_i) \leq k$. Therefore, by the induction hypothesis, (1) $\text{NAME}(\phi(w_i))$ passes $\text{NTEST}(w_i)$; (2) $\phi(w_i)$ satisfies $\text{PREDICATE}(w_i)$; and (3) $\phi(\text{LEAF}(w_i))$ belongs to $\text{SELECT}(\text{LEAF}(w_i)|w_i = \phi(w_i))$. By the axis match property of ϕ, we also have: (4) $\phi(w_i)$ relates to $\phi(w)$ according to $\text{AXIS}(w_i)$. From (1),(2),(4) we obtain that $\phi(w_i) \in \text{SELECT}(w_i|w = \phi(w))$. Combining this and (3), we have that $\phi(\text{LEAF}(w_i)) \in \text{SELECT}(\text{LEAF}(w_i)|w = \phi(w))$.

By the value match property of ϕ, $\text{STRVAL}(\phi(\text{LEAF}(w_i))) \in \text{TRUTH}(\text{LEAF}(w_i))$. We thus found a node in $\text{SELECT}(\text{LEAF}(w_i)|w = \phi(w))$ whose string value belongs to $\text{TRUTH}(\text{LEAF}(w_i))$. This implies that $\text{PEVAL}(r_{P_i}, \phi(w)) = \texttt{true}$ (Proposition A.3). Since this holds for all $i = 1, \ldots, \ell$, and $\text{PREDICATE}(u)$ is a conjunction of its atomic predicates, $\phi(w)$ satisfies $\text{PREDICATE}(w)$. $\square$

It follows from the above claim that $\phi(u)$ satisfies $\text{PREDICATE}(u)$, and thus the second condition of the lemma is met. We prove the third condition by induction on $\text{DIST}(u, v)$.

Assume, initially, that $\text{DIST}(u, v) = 0$. That is, $v = u$. In this case $\text{SELECT}(u|u = \phi(u)) = \{\phi(u)\}$, and therefore the condition is met. Suppose the condition holds for all nodes v, for which $\text{DIST}(u, v) \leq t$. We will show it holds also for nodes v with $\text{DIST}(u, v) = t + 1$.

Since $\text{DIST}(u, v) = t + 1$, then $\text{DIST}(u, \text{PARENT}(v)) \leq t$. Therefore, by the induction hypothesis, $\phi(\text{PARENT}(v)) \in \text{SELECT}(\text{PARENT}(v)|u = \phi(u))$. Note that: (1) $\phi(v)$ passes $\text{NTEST}(v)$ (by the node test match property of ϕ); (2) $\phi(v)$ satisfies $\text{PREDICATE}(v)$ (by Claim A.4); and (3) $\phi(v)$ relates to $\phi(\text{PARENT}(v))$ according to $\text{AXIS}(v)$ (by the axis match property of ϕ). Therefore, $\phi(v) \in \text{SELECT}(v|\text{PARENT}(v) = \phi(\text{PARENT}(v)))$, and hence $\phi(v) \in \text{SELECT}(v|u = \phi(u))$, as desired. $\square$

Proof of Lemma A.2. We prove the lemma by induction on HEIGHT(u). Suppose, initially, that HEIGHT(u) = 0. Therefore, u is a leaf, and in particular, LEAF(u) = u. Therefore, SELECT(LEAF(u)|$u = x$) = $\{x\}$, and the only y in SELECT(LEAF(u)|$u = x$) is x itself. Hence, it must be the case that STRVAL(x) $\in$ TRUTH(u).

Define: $\phi(u) = x$. It's easy to check that ϕ is a proper matching of x with u: (1) **root match:** follows from the definition; (2) **axis match:** since u is a leaf, follows trivially; (3) **node test match:** NAME(x) passes NTEST(u) (that's given); and (4) **value match:** STRVAL(x) $\in$ TRUTH(u) (see above).

Assume now that the lemma holds for all nodes of maximum height at most k. Let u be a node of maximum height $k+1$. Let v be the successor of u and let $w_1, \ldots, w_t$ be its predicate children. We will construct the matching ϕ of x with u by pasting together matchings of $w_1, \ldots, w_t$ and of v. First, define: $\phi(u) = x$.

Let $P_1, \ldots, P_t$ be the constituent atomic predicates of PREDICATE(u) and let $r_{P_1}, \ldots, r_{P_t}$ be their roots. Since x satisfies PREDICATE(u), then PEVAL(r_{P_i}, x) = `true` for all $i = 1, \ldots, t$. Fix any such i. If w_i is the variable at P_i, we know from Proposition A.3 that there exists a node $y_i \in$ SELECT(LEAF(w_i)|$u = x$), so that STRVAL(y_i) $\in$ TRUTH(LEAF(w_i)). It follows there exists a node $x_i \in D$, so that $y_i \in$ SELECT(LEAF(w_i)|$w_i = x_i$) and $x_i \in$ SELECT(w_i|$u = x$). We next present a matching ϕ_i of x_i with w_i.

Note that: (1) x_i satisfies PREDICATE(w_i) and its name passes NTEST(w_i) (because $x_i \in$ SELECT(w_i|$u = x$)); (2) $y_i \in$ SELECT(LEAF(w_i)|$w_i = x_i$) and STRVAL(y_i) $\in$ TRUTH(LEAF(w_i)) (see above); and (3) HEIGHT(w_i) $\leq k$. Hence, we can apply the induction hypothesis on w_i, x_i, y_i and obtain a matching ϕ_i of x_i with w_i, for which ϕ(LEAF(w_i)) = y_i.

Let's turn now to the successor v of u, if it exists. Note that: LEAF(v) = LEAF(u). Therefore, if $y \in$ SELECT(LEAF(u)|$u = x$), there exists a node $x' \in D$, so that $y \in$ SELECT(LEAF(u)|$v = x'$) and $x' \in$ SELECT(v|$u = x$). Note that: (1) x' satisfies PREDICATE(v) and its name passes NTEST(v) (follows from the fact $x' \in$ SELECT(v|$u = x$)); (2) STRVAL(y) $\in$ TRUTH(LEAF(u)) (that's given); and (3) HEIGHT(v) $\leq k$. Hence, we can apply the induction hypothesis on v, x', y, and obtain a matching ϕ' of x' with v, for which ϕ'(LEAF(u)) = y.

We now prove that ϕ, the combination of $\phi', \phi_1, \ldots, \phi_t$ and the definition $\phi(u) = x$, is a proper matching of x with u.

1. **Root match:** Follows from our definition $\phi(u) = x$.

2. **Axis match:** Take any $w \in Q_u$, $w \neq u$. If $w = v$, then $\phi(v) = x'$ relates to $\phi(u) = x$ according to AXIS(v), due to the fact $x' \in$ SELECT(v|$u = x$). If $w = w_i$ for some $i \in \{1, \ldots, t\}$, then $\phi(w_i) = x_i$ relates to $\phi(u) = x$ according to AXIS(w_i), due to the fact $x_i \in$ SELECT(w_i|$u = x$). In all other cases, the axis match property follows from the corresponding property of ϕ' and $\phi_1, \ldots, \phi_t$.

3. **Node test match:** Take any $w \in Q_u$. If $w = u$, then $\text{NAME}(\phi(u)) = \text{NAME}(x)$ passes $\text{NTEST}(u)$ (that's given). In all other cases, the node test match property follows from the corresponding property of ϕ' and $\phi_1, \ldots, \phi_t$.

4. **Value match:** Take any $w \in Q_u$. If $w = u$, then $\text{TRUTH}(u) = \mathcal{S}$, because u is an internal node. Therefore, $\text{STRVAL}(\phi(u)) \in \text{TRUTH}(u)$ trivially. In all other cases, the value match property follows from the corresponding property of ϕ' and $\phi_1, \ldots, \phi_t$.

Therefore, ϕ is indeed a proper matching and $\phi(\text{LEAF}(u)) = \phi'(\text{LEAF}(u)) = y$, as desired. $\square$

Lemma 5.19 (restated) *If Q is strongly subsumption-free, then it is also subsumption-free.*

Proof. Suppose, to reach a contradiction, that Q is strongly subsumption-free but is not subsumption-free. Therefore, there exists a node $u \in Q$ and a set of nodes $S \subseteq Q \setminus \{u\}$, so that

$$\text{MATCHES}(u) \subseteq \bigcup_{v \in S} \text{MATCHES}(v).$$

Since Q is strongly subsumption-free, we can define the "canonical document" D_c corresponding to Q (see Section 6.4). The "canonical mapping" ϕ_c is a matching of D_c with Q (Lemma 6.11). Therefore, $\langle D_c, \phi_c(u) \rangle \in \text{MATCHES}(v)$ for some $v \in S$. Hence, there exists some matching ϕ' of D_c and Q so that $\phi'(v) = \phi_c(u)$. Since $v \neq u$, $\phi' \neq \phi_c$. We thus found a non-canonical matching of D_c with Q, which is a contradiction to Lemma 6.15. $\square$

B Hybrid matchings

In this section we prove the statement about hybrid matchings:

Lemma 6.7 (restated) *Let Q be a univariate query, let D be a document, let ϕ be a matching of a node $x \in D$ with a node $u \in Q$, and let η be a matching of D with Q_{-u}. If x relates to $\eta(\text{PARENT}(u))$ according to $\text{AXIS}(u)$, then the hybrid mapping μ induced by ϕ and η is a matching of D with Q.*

Proof. We show that the hybrid mapping μ possesses the four properties of a matching:

1. **Root match:** $\text{ROOT}(Q)$ does not belong to the subtree Q_u, and therefore $\mu(\text{ROOT}(Q)) = \eta(\text{ROOT}(Q)) = \text{ROOT}(D)$.

2. **Axis match:** Let $v \in Q$, $v \neq \text{ROOT}(Q)$. If $v \notin Q_u$, then also $\text{PARENT}(v) \notin Q_u$. Thus, for such nodes $\mu(v) = \eta(v)$ and $\mu(\text{PARENT}(v)) = \eta(\text{PARENT}(v))$. The axis match property in this case follows from the corresponding property of η.

If PARENT$(v) \in Q_u$, then also $v \in Q_u$, and therefore $\mu(v) = \phi(v)$ and $\mu(\text{PARENT}(v)) = \phi(\text{PARENT}(v))$. The axis match property in this case follows from the corresponding property of ϕ.

Consider then the case that $v \in Q_u$ but PARENT$(v) \notin Q_u$. This can happen only if $v = u$. In this case $\mu(u) = \phi(u) = x$, while $\mu(\text{PARENT}(u)) = \eta(\text{PARENT}(u))$. Thus, the axis match property in this case follows from the assumption that x relates to $\eta(\text{PARENT}(u))$ according to AXIS(u).

3. **Node test match** Follows directly from the node test match properties of ϕ and η.

4. **Value match** Follows directly from the value match properties of ϕ and η. □

C   Document homomorphisms

In this section we prove the properties of document homomorphisms described in Section 6.1.

Lemma 6.2 (restated) *Let D, D' be two documents, let $x \in D$ and $x' \in D'$ be two nodes in these documents, and assume there is a homomorphism (resp., structural homomorphism) ξ from D_x to $D'_{x'}$. Let Q be a redundancy-free query, and suppose there is a matching (resp., structural matching) ϕ of x with a node $u \in Q$. Then, the mapping $\eta \stackrel{\text{def}}{=} \xi \circ \phi$ is a matching (resp., structural matching) of x' with u.*

Proof. We first prove the lemma for the case ξ is a structural homomorphism and ϕ is a structural matching. We show that η is a structural matching of x' with u:

1. **Root match:** $\eta(u) = \xi(\phi(u)) = \xi(x) = x'$.

2. **Axis match:** Let $v \in Q_u$, $v \neq u$. If v has a child axis (resp., descendant axis), then by the axis match property of ϕ, $\phi(v)$ is a child (resp., descendant) of $\phi(\text{PARENT}(v))$. By the axis preservation property of ξ, $\xi(\phi(v))$ is then a child (resp., descendant) of $\xi(\phi(\text{PARENT}(v))$.

3. **Node test match:** Let v be a node in Q_u whose node test is not $*$. By the node test match property of ϕ, NAME$(\phi(v)) = $ NTEST(v). By the node test preservation property of ξ, NAME$(\xi(\phi(v))) = $ NAME$(\phi(v)) = $ NTEST(v).

If ξ is now a homomorphism and ϕ is a matching, they are in particular a structural homomorphism and a structural matching, respectively. Therefore, by the above η has the first three properties of a matching. We are left to prove it has the value match property.

Let $v \in Q_u$. By the value match property of ϕ, STRVAL$(\phi(v)) \in$ TRUTH(v). By the value preservation property of ξ, STRVAL$(\eta(v)) =$ STRVAL$(\phi(v))$. □

Lemma 6.4 (restated) *Let D, D' be two documents, let $x \in D$ and $x' \in D'$ be two nodes in these documents, and assume there is a weak homomorphism ξ from D_x to $D'_{x'}$. Let Q be a redundancy-free query, and suppose there is a leaf-preserving matching ϕ of x with a node $u \in Q$. Then, the mapping $\eta \overset{\text{def}}{=} \xi \circ \phi$ is a matching of x' with u.*

Proof. ξ is a weak homomorphism from D_x to $D'_{x'}$, and therefore is in particular a structural homomorphism. Similarly, ϕ is a leaf-preserving matching of x with u, and therefore is in particular a structural matching. Hence, by Lemma 6.2, the mapping η already satisfies the first three properties of a matching. We are left to prove that η has the value match property.

Let v be any node in Q_u. If v is internal, then since Q is leaf-only-value-restricted, TRUTH$(v) = \mathcal{S}$, and thus the value match property of η follows trivially. So assume v is a leaf. Since ϕ is leaf-preserving, $\phi(v)$ is a leaf. By the value match property of ϕ, STRVAL$(\phi(v)) \in$ TRUTH(v). By the leaf value preservation property of ξ, STRVAL$(\xi(\phi(v))) =$ STRVAL$(\phi(v))$. □

D   Query automorphisms

In this section we prove the basic properties of structural query automorphisms discussed in Section 6.4.

Lemma 6.9 (restated) *A node $u \in Q$ structurally subsumes a node $v \in Q$ if and only if there exists a structural query automorphism ψ on Q, such that $\psi(v) = u$.*

Proof. Suppose, initially, that there exists a structural query automorphism ψ on Q, so that $\psi(v) = u$. We will prove that u must structurally subsume v.

Let $\langle D, x \rangle$ be any document-node pair in SMATCHES(u). That is, x is a node in D and structurally matches u. Let ϕ be the structural matching of D and Q with $\phi(u) = x$. We next define a new structural matching ϕ' of D and Q so that $\phi'(v) = x$. That would imply that $\langle D, x \rangle$ also belongs to SMATCHES(v), proving that u structurally subsumes v.

For any node $w \in Q$, we define $\phi'(w) = \phi(\psi(w))$. First note that $\phi'(v) = x$ as desired. We next prove that ϕ' is indeed a structural matching.

1. **Root match:** $\phi'(\text{ROOT}(Q)) = \phi(\psi(\text{ROOT}(Q))) = \phi(\text{ROOT}(Q)) = \text{ROOT}(D)$.

2. **Axis match:** Let $w \in Q$, $w \neq \text{ROOT}(Q)$. If w has a child axis, then $\psi(w)$ is a child of $\psi(\text{PARENT}(w))$ and $\psi(w)$ has a child axis. Therefore, $\phi'(w) = \phi(\psi(w))$ is a child of $\phi'(\text{PARENT}(w)) = \phi(\psi(\text{PARENT}(w)))$. If w has a descendant axis, then $\psi(w)$ is a descendant of $\psi(\text{PARENT}(w))$, and therefore also $\phi'(w) = \phi(\psi(w))$ is a descendant of $\phi'(\text{PARENT}(w)) = \phi(\psi(\text{PARENT}(w)))$.

3. **Node test match:** For any $w \in Q$, if $\text{NTEST}(w) \neq *$, then $\text{NTEST}(\psi(w)) = \text{NTEST}(w)$. That implies that also $\text{NAME}(\phi'(w)) = \text{NAME}(\phi(\psi(w))) = \text{NTEST}(w)$. Therefore, the name of $\phi'(w)$ passes the node test of w.

For the other direction, we assume that u structurally subsumes v. We need to show that there is a structural query automorphism ψ on Q, so that $\psi(v) = u$.

We use the construction of a "structurally canonical document" described in Section 6.4. Let D_c be such a document. The canonical matching ϕ_c of D_c and Q is also a structural matching of D_c and Q. Thus, the node $\phi_c(u) = \text{SHADOW}(u)$ structurally matches the node u. Since u structurally subsumes v, there is another structural matching ϕ of D_c and Q, such that $\phi(v) = \phi_c(u)$.

The structural matching ϕ induces a structural query automorphism ψ, as described in Lemma 6.14. Recall that $\psi(w) = \text{SHADOW}^{-1}(\phi(w))$, for all nodes $w \in Q$. Therefore, $\psi(v) = u$, as desired. $\qquad\square$

Proposition 6.10 (restated) *Let ψ be any structural query automorphism on Q. Then, for all $u \in Q$, $\text{DEPTH}(u) \leq \text{DEPTH}(\psi(u))$.*

Proof. We prove that $\text{DEPTH}(u) \leq \text{DEPTH}(\psi(u))$ by induction on $\text{DEPTH}(u)$. If $\text{DEPTH}(u) = 0$, then $u = \text{ROOT}(Q)$, and therefore $\psi(u) = \text{ROOT}(Q)$. We thus have $\text{DEPTH}(u) = 0 = \text{DEPTH}(\psi(u))$ in this case.

Assume, then, that $\text{DEPTH}(u) \leq \text{DEPTH}(\psi(u))$ for all u with $\text{DEPTH}(u) \leq k$. Let u be a node of depth $k + 1$. Therefore, $\text{PARENT}(u)$ is of depth k. By the induction hypothesis, $\text{DEPTH}(\text{PARENT}(u)) \leq \text{DEPTH}(\psi(\text{PARENT}(u)))$. By the axis preservation property of ψ, $\psi(u)$ is a descendant of $\psi(\text{PARENT}(u))$. Therefore, $\text{DEPTH}(\psi(u)) \geq \text{DEPTH}(\psi(\text{PARENT}(u))) + 1 \geq \text{DEPTH}(\text{PARENT}(u)) + 1 = \text{DEPTH}(u)$. $\qquad\square$

E   Canonical documents

In this section we prove statements about canonical documents discussed in Section 6.4

Proposition 6.16 (restated) *For any node $u \in Q$, no descendant of $\text{SHADOW}(u)$ has a matching with u.*

Proof. We prove the proposition by induction on the maximum height of the node u. When u is a leaf (maximum height 0), its shadow has no descendants, and therefore the claim follows trivially.

Suppose, then, that the claim holds for all nodes of maximum height at most d. Let u be a node of maximum height $d + 1$ and assume, to reach a contradiction, there exists a matching η of a descendant y of $\text{SHADOW}(u)$ with u.

We consider two cases: either $y = \text{SHADOW}(v)$ for some $v \in Q$, or y is artificial. Consider first the case $y = \text{SHADOW}(v)$. Since $\text{SHADOW}(v)$ is a descendant of $\text{SHADOW}(u)$, v has to be a descendant of u. Now, η restricted to Q_v is a matching of $\eta(v)$ with v. Since $\eta(v)$ is a descendant of $\eta(u) = y = \text{SHADOW}(v)$, we found a descendant of $\text{SHADOW}(v)$ that has a matching with v. This contradicts the induction hypothesis, because v is of maximum height at most d.

Consider now the case that y is an artificial node. This means that y belongs to a chain of $h + 1$ artificial nodes that ends with $\text{SHADOW}(v')$ for some $v' \in Q$ that has a descendant axis. Since $\text{SHADOW}(v')$ is a descendant of $\text{SHADOW}(u)$, v' has to be a descendant of u. The restriction of η to $Q_{v'}$ is a matching of $\eta(v')$ with v'. $\eta(v')$ is a descendant of $\eta(u) = y$ and therefore either (1) it belongs to the chain of artificial nodes connecting y and $\text{SHADOW}(v')$; or (2) it is a descendant of $\text{SHADOW}(v')$; or (3) equals $\text{SHADOW}(v')$.

Suppose that $\eta(v')$ is an artificial node. Then, the name of $\eta(v')$ is a the auxiliary name, and thus $\text{NTEST}(v')$ must be the wildcard. This is impossible, since $\text{AXIS}(v') = \texttt{descendant}$, and no node in Q has both a $\texttt{descendant}$ axis and a wildcard node test (recall that Q is star-restricted). So $\eta(v')$ cannot be an artificial node.

Suppose then that $\eta(v')$ is a descendant of $\text{SHADOW}(v')$. Then we found a node $v' \in Q$ whose shadow $\text{SHADOW}(v')$ has a descendant $\eta(v')$ that has a matching with v'. Since v' is of maximum height at most d, this contradicts the induction hypothesis.

We are left to address the case $\eta(v') = \text{SHADOW}(v')$. Let $u_1, \ldots, u_\ell$ be the path segment connecting u and v'; that is $u_1 = u$, $u_\ell = v'$, and u_j is a child of u_{j-1} for $j = 2, \ldots, \ell$. Let $y_1, \ldots, y_t$ be the path segment connecting y and $\text{SHADOW}(v')$. Note that $\eta(u_1) = y = y_1$ and $\eta(u_\ell) = \eta(v') = \text{SHADOW}(v') = y_t$.

y_1 is an artificial node and thus its name is the auxiliary name. This means that u_1 has a wildcard node test. Recall that Q is star-restricted, and therefore all the children of u_1, u_2 in particular, have a child axis. The only child of y_1 is y_2. Therefore $\eta(u_2) = y_2$. Let $k = \min(t, \ell)$. By applying the same argument inductively, we obtain that for $j = 2, \ldots, k-1$, u_j has a wildcard node test and for $j = 2, \ldots, k$, u_j has a child axis and $\eta(u_j) = y_j$.

Recall that $\eta(u_\ell) = y_t$. If $\ell < t$, then by the above $\eta(u_\ell) = y_\ell \neq y_t$. If $\ell > t$, then $\eta(u_t) = y_t$, and therefore $\eta(u_\ell)$ has to be a descendant of y_t. Thus the only way $\eta(u_\ell) = y_t$ is that $\ell = t = k$. But now $u_t = v'$ has a descendant axis and not a child axis. $\quad\square$

Lemma 6.19 (restated) *Let ξ be an internal node preserving weak homomorphism from a document D to D_c, and let ϕ be a matching of D and Q. Then, the mapping $\eta \overset{\text{def}}{=} \xi \circ \phi$ is a matching of D_c and Q (and thus equals the canonical matching ϕ_c).*

Proof. The idea is to show that ϕ must be leaf-preserving, and then resort to Lemma 6.4. Assume, to reach a contradiction, that ϕ is not leaf-preserving. Therefore, there exists a leaf node $v \in Q$, so that $\phi(v)$ is an internal node of D. Since ξ is internal node preserving, also $\eta(v)$ is an internal node of D_c.

Since ξ is in particular a structural homomorphism and ϕ is in particular a structural matching, then by Lemma 6.2, η is a structural matching of D_c and Q. Now, since $\eta(v)$ is not an artificial node (Lemma 6.12), it equals $\text{SHADOW}(u)$, for some internal node $u \in Q$. Lemma 6.14 now implies that u structurally subsumes v. That is, v is a leaf in the structural domination set of u.

It follows from the construction of D_c that $\text{SHADOW}(u)$ has a `text` node child, preceding all its other children, whose text content is a string α, which is not a prefix of any string in $\text{TRUTH}(v)$.

Now, recall that ξ maps $\phi(v)$ to $\text{SHADOW}(u)$ and that $\phi(v)$ is an internal node of D. Since ξ is internal node preserving, the `text` node child of $\phi(v)$ preceding its other children has the same text content as the `text` node child of $\xi(\phi(v))$ preceding its other children. We conclude that α is the prefix of $\text{STRVAL}(\phi(v))$. Since α is not the prefix of any value in $\text{TRUTH}(v)$, $\text{STRVAL}(\phi(v)) \notin \text{TRUTH}(v)$. This means that ϕ does not satisfy the value match property, and is therefore not a valid matching. We reached a contradiction. $\quad\square$