IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

The Space Complexity of Processing XML Twig Queries Over Indexed Documents

Mirit Shalem and Ziv Bar-Yossef

CCIT Report #693
March 2008

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #693 March 2008

The Space Complexity of Processing XML Twig Queries Over Indexed Documents*

Mirit Shalem
Department of Computer Science
Technion, Haifa, Israel
mirit2s@cs.technion.ac.il

Ziv Bar-Yossef
Department of Electrical Engineering
Technion, Haifa, Israel
and Google Haifa Engineering Center
Haifa, Israel
zivby@ee.technion.ac.il

March 27, 2008

Abstract

Current twig join algorithms incur high memory costs on queries that involve child-axis nodes. In this paper we provide an analytical explanation for this phenomenon. In a first large-scale study of the space complexity of evaluating XPath queries over indexed XML documents we show the space to depend on three factors: (1) whether the query is a path or a tree; (2) the types of axes occurring in the query and their occurrence pattern; and (3) the mode of query evaluation (filtering, full-fledged, or "pattern matching"). Our lower bounds imply that evaluation of a large class of queries that have child-axis nodes indeed requires large space.

Our study also reveals that on some queries there is a large gap between the space needed for pattern matching and the space needed for full-fledged evaluation or filtering. This implies that many existing twig join algorithms, which work in the pattern matching mode, incur significant space overhead. We present a new twig join algorithm that avoids this overhead. On certain queries our algorithm is exceedingly more space-efficient than existing algorithms, sometimes bringing the space down from linear in the document size to constant.

1 Introduction

XQuery and XPath [9] queries are typically represented as node-labeled *twig patterns* (i.e., small trees). Evaluating a twig pattern over an XML document is therefore a core database operation. As with relational databases, creating an index over the XML document at a pre-processing step can significantly reduce the costs (time, space) of query evaluation. Similarly to text search, an index for an XML document consists of *posting lists* or *streams*, one for each XML label that occurs in the document. The stream consists of positional encodings of all the elements that have this label, in document order. In this paper we focus on the most popular encoding scheme, the *BEL encoding* [6], in which each element is encoded as a (Begin,End,Level) tuple. The BEL encoding, although being compact, enables simple testing of structural relationships between elements.

Over the past decade, many algorithms for evaluating twig queries over indexed XML documents have been proposed (e.g., [6, 7, 10, 23, 22, 25, 18]). Much progress has been made in supporting wider fragments

*Supported by the European Commission Marie Curie International Re-integration Grant.

of XPath and XQuery and in achieving better performance in terms of running time, memory usage, and I/O costs.

Many of the existing algorithms follow two trends. The first trend is the tendency to achieve good performance on queries that involve descendant-axis only nodes, while suffering from poor performance on queries that involve child-axis nodes. The second trend relates to the mode of evaluation: many current algorithms find all possible matches of the *whole* query in the document ("pattern matching"), even though they are required to output only the matches of the query's *output node(s)* ("full-fledged evaluation") or to simply return a *bit* indicating whether there is at least one match of the query in the document ("filtering").

These trends raise two natural questions. First, is the space overhead incurred by child-axis nodes inherent or is it an artifact of the way existing algorithms work? Second, does the pattern matching evaluation mode incur any overhead relative to full-fledged evaluation and/or filtering?

Our results. In order to address the above questions, we embark on a large-scale study of the space complexity of evaluating twig queries over indexed documents. Our lower bound results are quite strong, since they apply to the *instance data complexity* [5], rather than to the standard *data complexity*. That is, we fix *any* query, not just a worst-case query, and then prove lower bounds for evaluating this query. Therefore, our lower bounds are given in terms of properties of the query as well as parameters of the document. Our analysis shows that the space complexity of twig query evaluation depends on three parameters: (i) whether the query is a path or a tree; (ii) the types of the axes in the query and their occurrence pattern; and (iii) the mode of evaluation: filtering, full-fledged evaluation, or pattern matching.

Filtering				
Axis pattern	Path queries		Tree queries	
	Upper bound	Lower bound	Upper bound	Lower bound
$(/)^*(//)^*$	$\tilde{O}(1)$	$\Omega(1)$	$\tilde{O}(1)$	$\Omega(1)$
$(//\|/)^*(//)(/)(//\|/)^*$	$\tilde{O}(d)$	$\Omega(d)$	$\tilde{O}(d)$	$\Omega(d)$

Full-fledged evaluation				
Axis pattern	Path queries		Tree queries	
	Upper bound	Lower bound	Upper bound	Lower bound
$(/)^*(//)^*$	$\tilde{O}(1)$	$\Omega(1)$	$\tilde{O}(1)$	$\Omega(1)$
$(//\|/)^*(//)(/)(//\|/)^*$	$\tilde{O}(d)$	$\Omega(d)$	$\tilde{O}(D)$	$\Omega(\max(d,out))$

Pattern Matching				
Axis pattern	Path queries		Tree queries	
	Upper bound	Lower bound	Upper bound	Lower bound
$(/)^*(//)?$	$\tilde{O}(d)$	$\Omega(1)$	$\tilde{O}(out)$	$\Omega(out)$
$(/)^*(//)(//)^+$	$\tilde{O}(d)$	$\Omega(d)$	$\tilde{O}(out)$	$\Omega(out)$
$(//\|/)^*(//)(/)(//\|/)^*$	$\tilde{O}(d)$	$\Omega(d)$	$\tilde{O}(D)$	$\Omega(\max(d,out))$

Table 1: Summary of our results. D, d, and "out" denote the document size, its depth, and the output size, resp.

Table 1 summarizes our results (marked in shaded background) as well as previously known bounds. We analyze each evaluation mode separately. We also categorize the queries according to the axis pattern of paths in the query (represented as a regular expression). To classify a query, we check if at least one path in the query fits the regular expression, starting from the lowest row of the table upwards. As the query size is typically small relative to the document size or the output size, we did not focus on it as a parameter. We use the $\tilde{O}$ notation to suppress factors that are linear in the query size or logarithmic in the document size.

Our results provide two theoretical explanations for the difficulty in handling queries with child-axis nodes. The first explanation applies to all evaluation modes and to queries that contain the (//)(/) pattern (i.e., ones that consist of at least one descendant-axis node that is followed by a child-axis node, such as //*a*/*b*; see the last row in all three tables). We show that the space needed to evaluate such queries is $\Omega(d)$, where d is the document's depth. The lower bound follows from the need to simultaneously hold in memory candidate matches of the descendant-axis node that are nested within each other. Thus, when evaluating the query on highly recursive documents (ones that consist of long chains of same-label elements), $\Omega(d)$ space may be needed. The second, and possibly more significant, explanation applies to the full-fledged evaluation and pattern matching modes and to (a subset of) the tree queries that contain the (//)(/) pattern (see the lower right corner at the second and third tables). We prove that processing such queries additionally requires $\Omega(\text{out})$ space, where "out" is the output size (approximately, the number of matches of the query in the document). As the output size can be as large as the document itself, it may be unavoidable to use a lot of memory on such queries.[1]

Our study reveals another notable phenomenon. On tree queries that do not contain the (//)(/) pattern (i.e., ones that consist of descendant-axis nodes only or ones in which child-axis nodes always precede descendant-axis nodes; see the upper right corners in all three tables), pattern matching is subject to an $\Omega(\text{out})$ lower bound, while the other modes are not. We present a new twig join algorithm that is adapted for the filtering and full-fledged evaluation modes and uses only *constant* space for these queries. Thus, our algorithm demonstrates that working in the pattern matching mode, while only filtering or full-fledged evaluation are needed, incurs significant space overhead.

The $\tilde{O}(d)$ upper bound for path queries (in all evaluation modes) follows from the PathStack algorithm [6]. The $\tilde{O}(d)$ upper bound for filtering tree queries follows from the TurboXPath algorithm [20][2] (see also [18, 5]). The tight upper bounds in the pattern matching mode (see the second row of the third table) follow from extensions we propose to the TwigStack algorithm [6]. Obtaining space-optimal algorithms for tree queries that contain the (//)(/) pattern remains an open problem. ($\tilde{O}(D)$ is the space needed by an in-memory algorithm that simply stores the whole document in main memory.)

This paper focuses on space complexity. Analysis of other complexity measures, such as running time and I/O is postponed to future work.

Our techniques. Space lower bounds for data stream algorithms are normally proved via communication complexity. It turns, however, that the standard communication complexity model [21] is inadequate for proving lower bounds for *multiple data stream* (MDS) algorithms. We therefore introduce a new model of multi-party communication complexity—the *token-based mesh communication model* (TMC)—which enables proving space lower bounds for MDS algorithms. The model allows a clean abstraction of the information-theoretic arguments made in the lower bound proofs. It also enables us to recycle arguments that are repeatedly used in the proofs, thus making them more modular. We prove communication lower bounds in the TMC model for two variants of the *set-disjointness* problem and for the *tensor product* problem. Our space lower bounds for twig query evaluation are obtained via reductions from these problems.

[1] Note that in general the algorithm does not need to allocate expensive main memory storage for the output, since the output can be written to a write-once output device.

[2] TurboXPath is designed for XML streams, yet it can be made to work on indexed XML documents with a constant factor space overhead.

Our new twig join algorithm differs substantially from previous approaches. Like TwigStack and its successors, our algorithm is "holistic", as it treats the whole query as one unit. Yet, unlike TwigStack, our algorithm is not "document-driven", but rather "query-driven". That is, rather than traversing the elements in document order and at each step looking for the largest query subtree that is matched by the current document subtree, our algorithm traverses the query top-down and advances the stream cursors to the next match. We include detailed theoretical correctness and performance analysis of our algorithm. As the main thrust of this paper is the analytical study of the space complexity of processing twig queries, we do not include empirical analysis of our algorithm. This is left for future work.

Outline. We begin by presenting background information, and a formal definition of the evaluation model. In Section 4 we prove our lower bounds, and describe the generic reduction scheme we use in our proofs. Section 5 presents the TMC model and communication lower bounds for three problems. In Section 6 we discuss existing and new upper bounds, and in particular, we present our twig join algorithm with its full analysis.

2 Related work

Starting with the seminal work of Bruno *et al.* [6] on holistic twig join algorithms, there have been many follow-up studies that presented improvements in the I/O and memory costs (e.g., [19, 23, 22, 10]) or extended the supported fragment of queries (e.g., [18, 25]). However, none of these papers presents a systematic study of lower bounds as we do.

The only previous work to address space lower bounds for processing twig queries was a paper by Choi *et al.* [8]. They state that any algorithm evaluating the query //a[a and a] requires super-constant memory.[3] Our study does not address a single worst-case query, but provides lower bounds for evaluation of *any* query. Our lower bounds are also finer-grained and yield a quantitative characterization of the space complexity.

Chen *et al.* [7] compared three different indexing schemes: by label, by label and level, and by ancestors' labels. They demonstrate the impact of the chosen scheme on the classes of twig patterns that can be evaluated "optimally", i.e., without redundant intermediate results. However, their focus is on the two latter schemes, and not on lower bounds for the first scheme, which is the subject of study in this paper.

Several previous works proved space lower bound for evaluating XPath queries in other models. Gottlob, Koch, and Pichler [12], Segoufin [24], and Götz, Koch and Martens [13] studied the complexity of evaluating XPath queries over XML documents stored in main memory. Grohe, Koch, and Schweikardt [16] proved lower bounds for XPath evaluation on external memory machines with limited random accesses. As the models studied in these works are completely different from the model studied in this paper, their lower bounds are not applicable to our setting. Bar-Yossef, Fontoura, and Josifovski [5, 4] showed space lower bounds for evaluating XPath queries over a *single* XML stream. Lower bounds in our model derive the same lower bounds in the their model, while upper bounds in their model also apply in our model.

There is extensive literature on massive data sets computations in general, and on multiple data streams in particular. Various models have been presented and analyzed, e.g. [3, 2, 17, 16, 15, 11, 1]. All these models are different from the multiple data stream model studied in this paper, either because they are stronger (and thus admit *weaker* lower bounds) or because they focus on other complexity measures than space.

The multiple-cursor multiple data stream model was analyzed in [14] and a lower bound for reverse set-disjointness was provided. Yet, the paper focuses on relational algebra queries and not on XPath.

[3] While this statement is true, we suspect the proof included in [8] to be flawed, as it relies on a reduction to, and not from, evaluation of Select-Project-Join queries over continuous data streams [2].

3 Preliminaries

Data model. XML documents are modeled as ordered rooted trees. Each node in the tree is called an *element* and is labeled by a name or a text value. The edges represent direct element-subelement or element-value relationships. Every document has an (invisible) root whose label we denote by "$". Figure 1 depicts an example document tree.

Similarly to previous papers on twig joins, we assume only leaf elements in the document may contain text. This makes the relationship element-value easier to represent and evaluate.

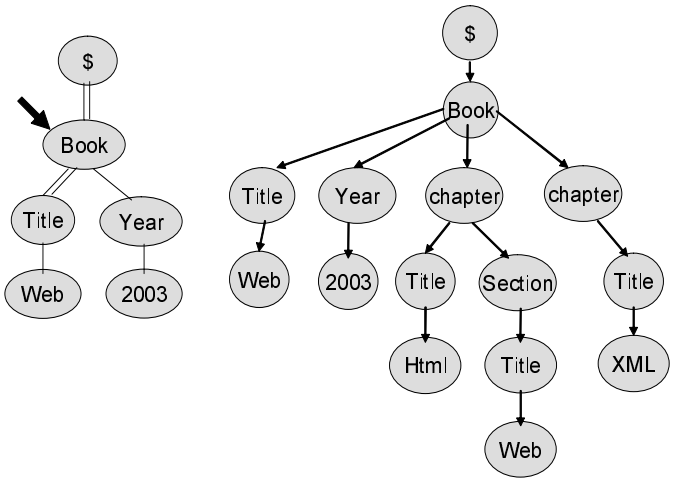

Figure 1: Example XML document (right) and twig query (left) for the XPath query: //$Book$[.//$Title$="Web" and $Year$=2003].

XPath fragment. We focus on a fragment of XPath, which we call *basic twig queries*. Many existing algorithms focus on this type of queries [6, 22, 19]. The syntax of a basic twig query is defined as follows :
$Twig ::= Step \mid Step\, Twig$
$Step ::= Node$ [$Predicate$]?
$Path ::= Node \mid Node\ Path$
$Node ::= (/|//)\ label$
$Predicate ::= Twig \mid Path$ = $textvalue \mid Predicate$ and $Predicate$

A basic twig query can be represented as a tree, where each internal node is marked by a label and each leaf is marked by a label or by a text value. Similarly to documents, every query has an invisible root labeled by "$". One of the tree's nodes is designated as the output node[4]. Figure 1 depicts an example basic twig query. The output node is pointed by an arrow.

Evaluation model. We consider query evaluation over *indexed XML documents*. An XML document is represented in *positional encoding*. Each document node is encoded as a triple: (Begin, End, Level), based on its position in the document. "Begin" and "End" are the positions of the beginning and the end of the element, respectively, and "Level" is the nesting depth. Positional encoding is the most popular format for representing XML documents, since it is simple and compact, yet it allows for efficient evaluation of structural relationships between document nodes.

[4]The output node in an XPath query is always the path's leaf. Yet, in the tree representation, this leaf may become any labeled node in the tree. For example, the two queries //a[b and c] and //a[b]/c are represented by the same tree, but their output nodes are different.

An indexed XML document consists of a collection of index streams, one stream for every label that occurs in the document. For every label 'a', stream T_a contains positional encodings of all elements with label 'a' in the document, sorted by the "Begin" attribute. Each query node u is associated with a cursor in the corresponding stream T_u. An algorithm can read from a cursor position many times, until it decides to advance it. Cursors can be advanced only forwards, and not backwards. The output is written to a write-only stream. If two query nodes u, v share the same label, the algorithm maintains two separate cursors on streams T_u and T_v, which represent the same stream. We therefore abuse notation and use T_u to denote the *cursor* on the stream corresponding to u. As mentioned earlier, the algorithms we consider are restricted to access only streams corresponding to labels that occur in the query. All known twig join algorithms conform to this restriction.

When analyzing the space complexity of an algorithm that runs over an indexed XML document, we do not take into account the space used for storing the input streams, the cursors, or the output stream.

Modes of evaluation. We consider three modes of query evaluation: *filtering*, *full-fledged*, and *pattern matching*. The underlying notion in all modes is a *match*.

For a query Q and a node $u \in T$, we denote by Q_u the sub-query rooted at u. Similarly, for a document D and an element $e \in D$, we denote by D_e the sub-document rooted at e.

Definition 3.1 (Sub-query match)**.** A *match of a sub-query Q_u in a sub-document D_{e_u}* is a mapping ϕ from the nodes of Q_u to elements in D_{e_u} satisfying the following: (1) **root match:** $\phi(u) = e_u$, (2) **labels match:** w and $\phi(w)$ have the same label, for every $w \in Q_u$, and (3) **structural match:** the structural relationship between $\phi(w)$ and $\phi(\mathrm{parent}(w))$ matches the axis of w, for every $w \in Q$, $w \neq u$.

A *match* of a query Q in a document D is a match of $Q_{\mathrm{root}(Q)}$ in $D_{\mathrm{root}(D)}$. Given a query Q and a document D, the *filtering* of D using Q, denoted $\textsc{filter}_Q(D)$, is a bit indicating whether Q has at least one match in D. The *pattern matching* of Q in D, denoted $\textsc{pm}_Q(D)$, is the collection of all matches of Q in D. The *full-fledged evaluation* of Q on D, denoted $\textsc{ffe}_Q(D)$, is the collection of elements $\phi(t)$, for all matches ϕ of Q in D (t is the output node of Q).

4 Lower bounds

In this section we present our lower bounds for evaluating basic twig queries, for the three evaluation modes. The proof of each lower bound is based on a reduction from another problem, whose lower bound we prove separately in Section 5. We start by describing our generic reduction scheme, and then we use it to prove three lower bounds, one for each evaluation mode.

4.1 Techniques

Our lower bounds are proved via reductions from problems in the *multiple data streams* (MDS) model:

The MDS model. In the multiple data streams (MDS) model, the input data x is divided into several read-only streams, and the required output, $f(x)$, is written to a write-only output stream. Each of the input streams is associated with a cursor that can move only in the forward direction. The cursor specifies which part of the stream has already been read. An algorithm can read from a cursor position many times, until it decides to advance it. When the entire input has been read, the output stream contains $f(x)$. This model generalizes our evaluation model for basic twig queries.

Definition 4.1 (MDSS)**.** The MDS *space complexity* of function f, denoted as MDSS(f), is the minimum space required for A, over all algorithms A that compute f in the MDS model.

Definition 4.2 (DSS)**.** Let DSS(f) denote the *(single) data stream space complexity* of function f, where f is a function whose input is given in *one* stream, i.e., DSS(f) = MDSS(f).

An MDS-reduction. Let f be a function in the MDS model over k streams: $s_1, \ldots, s_k$, and let g be a function in the MDS model over k+l streams: $t_1, \ldots, t_k$ and $c_1, \ldots, c_l$. We say $r = (r^1_{in}, \ldots, r^k_{in}, r^1_c, \ldots, r^l_c, r_{out})$ is an MDS reduction from f to g, denoted $f \leq_{\text{MDS}} g$, if r satisfies the following:

1. $\forall i, 1 \leq i \leq k : r^i_{in} : s_i \rightarrow t_i$
2. $\forall i, 1 \leq i \leq l : r^i_c : \epsilon \rightarrow c_i$ (the input is empty)
3. $r_{out} : Outputs(g) \rightarrow Outputs(f)$
4. $\forall (s_1, \ldots, s_k) \in Inputs(f) : r_{out}(g(r^1_{in}(s_1), \ldots, r^k_{in}(s_k), r^1_c(\epsilon), \ldots, r^l_c(\epsilon))) = f(s_1, \ldots, s_k)$

Lemma 4.3. *Suppose there exists an* MDS *reduction $r = (r^1_{in}, \ldots, r^k_{in}, r^1_c, \ldots, r^l_c, r_{out})$ from f to g. Then:* $\text{MDSS}(f) \leq \text{MDSS}(g) + \Sigma^k_{i=1}\text{DSS}(r^i_{in}) + \Sigma^l_{i=1}\text{DSS}(r^i_c) + \text{DSS}(r_{out})$

Proof. Assume the space optimal algorithms for $g, r^1_{in}, \ldots, r^k_{in}, r^1_c, \ldots, r^l_c$, and r_{out}, are $Ag, Ar^1_{in}, \ldots, Ar^k_{in}$, $Ar^1_c, \ldots, Ar^l_c$, and Ar_{out}, respectively. We now build an algorithm Af for f, whose space complexity is exactly $\text{MDSS}(g) + \Sigma^k_{i=1}\text{DSS}(r^i_{in}) + \Sigma^l_{i=1}\text{DSS}(r^i_c) + \text{DSS}(r_{out})$, by computing:

$$r_{out}(g(r^1_{in}(s_1), \ldots, r^k_{in}(s_k), r^1_c(\epsilon), \ldots, r^l_c(\epsilon)))$$

Specifically, we simulate Ar_{out}, whose input is the output stream of g. Therefore, whenever Ar_{out} advances the cursor of the input stream and reads the next bit, we simulate Ag until it outputs the next bit. But in order to simulate Ag, we need to generate on-the-fly its input streams: $t_1, \ldots, t_k$ and $c_1, \ldots, c_l$ (each time we generate the next requested bit). Therefore, whenever Ag advances the cursor of some input stream, say t_i, and reads its next bit, we simulate Ar^i_{in} until it outputs the next bit. Note that the input stream of Ar^i_{in} is s_i, which is given as the input for Af. If Ag advances the cursor of a c_i input stream, then we simulate Ar^i_c, whose input is ϵ.

The algorithm Af we described correctly computes f (by the MDS reduction properties), and its space complexity is $\text{MDSS}(g) + \Sigma^k_{i=1}\text{DSS}(r^i_{in}) + \Sigma^l_{i=1}\text{DSS}(r^i_c) + \text{DSS}(r_{out})$, since we simulate simultaneously all the corresponding algorithms. □

Corollary 4.4. *Suppose there exists an* MDS *reduction from f to g. Then:* $\text{MDSS}(g) \geq \text{MDSS}(f) - \Sigma^k_{i=1}\text{DSS}(r^i_{in}) - \Sigma^l_{i=1}\text{DSS}(r^i_c) - \text{DSS}(r_{out})$

4.2 Filtering

In this section we prove an Ω(docDepth) lower bound for filtering mode, for any query that contains the (//)(/) pattern (e.g., //a/b). Since filtering is easier than the other two modes of evaluation, the Ω(docDepth) lower bound applies also to both full-fledged evaluation and pattern matching of queries that consist of the (//)(/) pattern. This lower bound is matched by PathStack [6] for pattern matching of path queries, and by TurboXPath [20] for filtering any query.

Theorem 4.5. *Let a,b be any two labels, and let Q be any basic twig query that contains the path segment //a/b. Furthermore, assume $a \neq b$ and that a,b do not appear elsewhere in Q. Then, for every algorithm for* FILTER_Q *and for every $d \geq 1$, there exists a document of depth at most $d - 1 + \text{depth}(Q)$, on which the algorithm uses at least $d - O(|Q| \log(|Q| \cdot d))$ bits of space.*

The requirement that the labels a, b are different and unique in Q is made for technical reasons. We conjecture that the lower bound holds even if this requirement is not met. However, proving this would require a lower bound for a variant of the the reverse-set-disjointness problem (see below).

We first prove the theorem for the special case $Q = //a/b$. This proof captures the main technical challenges of the general case. We then present the proof of the general case.

The difficulty in finding whether the query $//a/b$ has a match or not emanates from recursive documents that contain multiple nodes named 'a' nested within each other. Any filtering algorithm will have to match the list of nested 'a' elements read from the stream T_a against the list of 'b' elements read from the stream T_b. The query has a match if and only if one of these 'b' elements is a child of an 'a' element. If for every 'a' element, the candidate 'b' child appears *after* the nested 'a' child/descendant, then due to the pre-order organization of elements in the two streams, T_a has to be matched against the *reverse* of T_b. This implies that no algorithm can match the two streams on the fly, but rather has to store at least one of them in memory. As the length of these streams can be as long as half of the document's depth, this implies the space lower bound.

Formally, the theorem is proven by an MDS reduction from the problem of *reverse-set-disjointness* in the MDS model.

Reverse set disjointness. Given two binary vectors: $x, y \in \{0,1\}^n$, the *reverse set disjointness* function, $\text{RDISJ}_n(x, y)$, is defined to be 1 if $\exists i$, s.t. $x_i = y_i^R = 1$, and 0 otherwise. Here, y^R is the reverse of y. When computed in the MDS model, x is given on one stream and y on another stream.

Theorem 4.6. *For any $n \geq 7$, the space complexity of RDISJ_n in the MDS model is at least $n - \log(n+1) - 3$.*

The proof appears in Section 5.2.

The reduction. We prove Theorem 4.5 for the case $Q = //a/b$ by an MDS reduction from RDISJ_n to FILTER_Q. Note that FILTER_Q can also be described as a function over two stream: $\text{FILTER}_Q(T_a, T_b)$, as we restricted ourselves to algorithms that access only streams corresponding to labels that occur in the query (see Section 3). Let $n = d - 1$. The MDS reduction is based on the following functions:

- r_{in}^1 and r_{in}^2 construct the index streams T_a and T_b, respectively, of the XML document $D(x, y)$ (see Figure 2). Specifically, $r_{in}^1(x) = T_a$ and $r_{in}^2(y) = T_b$. $D(x, y)$ is the same for all (x, y), except for the labeling of elements. When $x_i = 1$, the corresponding element s_i is labeled 'a', and otherwise it is labeled 'c'. When $y_i = 1$, the corresponding element t_i is labeled 'b', and otherwise it is labeled 'd'.
- $r_{out}(b) = b$ (note that the output of both RDISJ_n and FILTER_Q is one bit). Therefore $\text{DSS}(r_{out}) = 0$.

Claim 4.7. $\text{DSS}(r_{in}^1) = \text{DSS}(r_{in}^2) \leq \log n$.

Proof. We describe algorithms A and B for r_{in}^1 and r_{in}^2, resp. In order to output the next tuple in T_a [T_b], A [B] advances the stream x [y] to the next set bit. If the position of this bit is i, the algorithm creates the tuple $(i, 4n - 3i + 3, i)$ in T_a [$(n + 3i - 2, n + 3i - 1, n + 2 - i)$ in T_b]. If no set bit is found in x [y], the algorithm creates a T_a.End [T_b.End] tuple.

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the "Begin" attribute, and that they represent the document $D(x, y)$ whose depth is d. The space needed for A [B] is $\log n$ bits for keeping the current position in x [y]. □

Lemma 4.8. $\text{RDISJ}_n(x, y) = \text{FILTER}_Q(D(x, y))$.

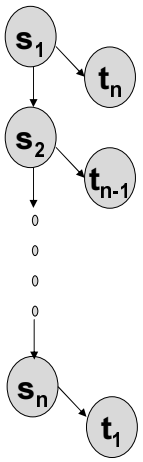

Figure 2: The document $D(x, y)$.

Proof. $\text{RDISJ}_n(x, y) = 1$ if and only if there exists some index $1 \leq i \leq n$, such that both x_i and y_i^R (i.e., y_{n+1-i}) are 1. This means that in $D(x, y)$ the label of s_i is 'a' and the label of t_{n+1-i} is 'b'. Since t_{n+1-i} is a child of s_i, the latter happens iff there is a match of Q in $D(x, y)$. □

Since r_{in}^1 and r_{in}^2 construct the index streams T_a and T_b of $D(x, y)$, it follows that:

$$r_{out}(\text{FILTER}_Q(r_{in}^1(x), r_{in}^2(y))) = \text{FILTER}_Q(r_{in}^1(x), r_{in}^2(y)) = \text{RDISJ}_n(x, y)$$

Therefore, by Corollary 4.4, $\text{MDSS}(\text{FILTER}_Q) \geq \text{MDSS}(\text{RDISJ}_n) - \text{DSS}(r_{in}^1) - \text{DSS}(r_{in}^2) - \text{DSS}(r_{out}) \geq n - O(\log n) = d - O(\log d)$.

Now that we have proved Theorem 4.5 for the special case $Q =$ *//a/b*, we present a proof sketch of the general case. A full proof of the theorem appears in Appendix A.

Proof sketch of Theorem 4.5. Similarly to the proof of the special case, the functions r_{in}^1 and r_{in}^2 construct the index streams T_a and T_b, respectively, of an XML document. The document's labels, as before, depend on x and y. However, since now the query is some arbitrary tree that contains the path segment *//a/b*, the document structure should match the query tree except, maybe, for the segment *//a/b*. This means that the document structure will resemble that of the special case (see Figure 2, but we add all the other query nodes "around" these potentially a (s_i) and b (t_i) nodes. This way the existence of a match of Q in this document depends only on the labels of s_i and t_i nodes, i.e., on the set bits in x and y. Note that since the query may consist of additional labels, the reduction should build their index streams too. However, these index streams do not depend on x, y, and can be "encoded" in the reduction. □

Remark (Regarding the case $a = b$). Theorem 4.5 required that the labels a, b are different and unique in Q. We noted earlier that if this requirement is not met, then proving the filtering lower bound requires a lower bound for a variant of the reverse-set-disjointness problem. The reason is that now the streams T_a and T_b are the same, i.e., the cursors of both a and b point to the same stream. The corresponding variant of the reverse-set-disjointness problem (in order to use the same MDS reduction shown above) is where both streams contain $x \circ y$.

4.3 Full-fledged evaluation

The $\Omega(\text{docDepth})$ lower bound we proved for filtering applies also to full-fledged evaluation of queries that consist of the (//)(/) pattern. In this section we prove that full-fledged evaluation of some of these queries is subject to an additional $\Omega(\text{outputSize})$ lower bound. Strictly speaking, the lower bound does not apply to all queries that contain (//)(/), but rather only to a subset of them, depending on the location of the output node in the query. These two lower bounds are combined to an $\Omega(\max(\text{docDepth}, \text{outputSize}))$ lower bound for this fragment of queries, as shown in Table 1 (see the middle table, lower right corner).

We define the *output size* of the evaluation of a query Q on a document D to be $|\text{FFE}_Q(D)|$, that is, the number of document nodes to which the query's output node can be matched.

Theorem 4.9 (Output size lower bound)**.** *Let Q be any basic twig query that contains the path segment //z/b. Furthermore, assume the following: (1) the output node is a descendant of the node labeled z but not of the node labeled b (this is where we require Q to be a tree and not a path); (2) the output node's label and z, b are distinct and do not appear elsewhere in Q. Then, for every algorithm for FFE_Q and for every $S \geq 1$, there exists a document D, for which $|\text{FFE}_Q(D)| \leq S$ and on which the algorithm uses at least $\Omega(S)$ bits of space.*

Here, we prove the theorem for the special case $Q = //z[b]/a$. This proof captures the main technical challenges of the general case. Formally, the theorem is proven by an MDS reduction from the following variant of the set-disjointness problem, which we call *delayed intersection*:

Delayed intersection. Given three binary vectors $s, t, u \in \{0,1\}^n$ and a bit $v \in \{0,1\}$, the *delayed intersection* function, $\text{DINT}_n(s,t,u,v)$, is defined as $(s \cap v^n) \circ (t \cap u)$, where $\cap$ denotes bitwise-and, v^n is the n-dimensional vector obtained by taking n copies of v, and $\circ$ denotes concatenation of vectors. For example, $\text{DINT}_n(101,011,101,1)$ is 101001. When computed in the MDS model, $s \circ t$ is given on one stream and $u \circ v$ on another stream.

Theorem 4.10. *For any $n \geq 7$, the space complexity of DINT_n in the MDS model is at least $n - \log(n+1) - 3$.*

The proof appears in Section 5.3.

The reduction. We prove Theorem 4.9 for the case $Q = //z[b]/a$ by an MDS reduction from DINT_n to FFE_Q. Let $n = S/2$. The MDS reduction is based on the following functions:

- $r^1_{in}(s \circ t)$ and $r^2_{in}(u \circ v)$ construct the index streams T_a and T_b, respectively, of an XML document $G(s,t,u,v)$. The document structure, which is presented in Figure 3, The document structure is the same for all inputs (s,t,u,v), except for the labeling of elements. When $s_i = 1$ or $t_i = 1$, the corresponding element s_i or t_i is labeled 'a', and otherwise it is labeled 'c'. When $u_i = 1$ or $v = 1$, the corresponding element u_i or v_0 is labeled 'b', and otherwise it is labeled 'd'.

- r^1_c constructs the index stream T_z of the document $G(s,t,u,v)$. Note that T_z is fixed and does not depend on the input.

- r_{out}: its input is a stream of a-elements from the document G, and it constructs a $2n$-bit vector, whose set bits correspond to the position of the a-elements. Specifically, an a element may be an (s_i) or a (t_i) node, and accordingly the position of the set bit is i or $n+i$, respectively. The position i may be computed from the element's tuple, and therefore $\text{DSS}(r_{out}) \leq \log n$.

Claim 4.11. $\text{DSS}(r^1_{in}) = \text{DSS}(r^2_{in}) \leq \log n$.

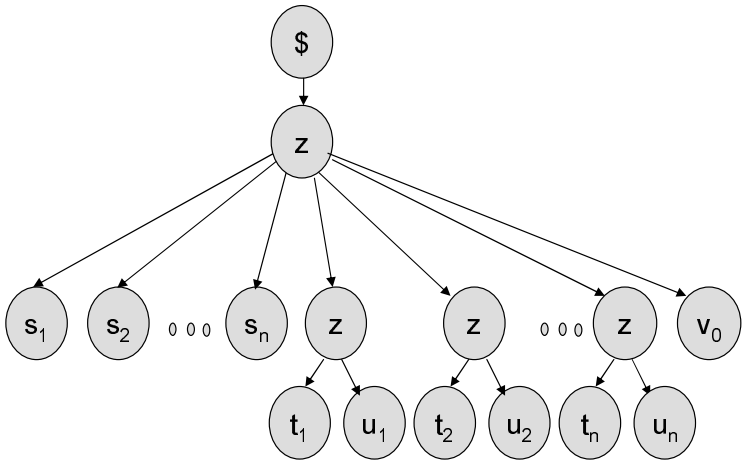

Figure 3: The document $G(s,t,u,v)$.

Proof. We describe algorithms A and B for r^1_{in} and r^2_{in}, resp. In order to output the next tuple in T_a [T_b], A [B] advances the stream $s \circ t$ [$u \circ v$] to the next set bit. The tuple created is a simple function of the position of this bit.

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the "Begin" attribute, and that they represent T_a and T_b of the document $G(s,t,u,v)$. The space needed for A [B] is $\log n$ bits for keeping the current position in the input stream $s \circ t$ [$u \circ v$]. □

Claim 4.12. $\text{DSS}(r^1_c) \leq \log n)$

Proof. The index stream T_z can be generated on-the-fly, based on the position of the required tuple. Therefore, only $\log n$ bits of space are needed, to maintain this position. □

The following proposition proves the validity of the reduction:

Proposition 4.13. *Let $1 \leq k \leq 2n$. The k-th bit in $\text{DINT}_n(s,t,u,v)$ is set iff:*
(i) if $k \leq n$, $s_k \in \text{FFE}_Q(G(s,t,u,v))$, and (ii) if $k > n$, $t_{k-n} \in \text{FFE}_Q(G(s,t,u,v))$.

Proof. If $k \leq n$, then the k-th bit is in $(s \cap v^n)$, and it is set iff both s_k and v are set. This means that the labels of elements s_k and v_0 in the document G are 'a' and 'b', respectively. The latter happens if and only if the mapping $\phi = (a \mapsto s_k, b \mapsto v_0)$ is a match of Q in G. If ϕ is a match, then $s_k \in \text{FFE}_Q(G(s,t,u,v))$. We only have to prove now that if $s_k \in \text{FFE}_Q(G(s,t,u,v))$, then ϕ is a match, i.e., prove that the labels of elements s_k and v_0 are 'a' and 'b', respectively. Since s_k was output, then there is a match that maps $a \mapsto s_k$. As the only element in G that is a child of s_k's parent and may have a 'b' label is v_0, the only possible match is ϕ.

The second case is when $k > n$. Now the corresponding bit is the $(k-n)$-th bit in $(t \cap u)$, which is set iff both t_{k-n} and u_{k-n} are set. The proof here is similar to the previous case, but with elements t_{k-n} and u_{k-n} instead of s_k and v_0. □

Since $(r^1_{in}, r^2_{in}, r^1_c)$ construct the index streams of $G(s,t,u,v)$, it follows that:

$$r_{out}(\text{FFE}_Q(r^1_{in}(s \circ t), r^2_{in}(u \circ v), r^1_c(\epsilon))) = \text{DINT}_n(s,t,u,v)$$

Therefore, by Corollary 4.4,

$$\text{MDSS}(\text{FFE}_Q) \geq \text{MDSS}(\text{DINT}_n) - \text{DSS}(r^1_{in}) - \text{DSS}(r^2_{in}) - \text{DSS}(r^1_c) - \text{DSS}(r_{out}) \geq n - O(\log n)) = \Omega(s)$$

Note also that the output size of $\text{FFE}_Q(G(s,t,u,v))$ is at most $2n = S$. This concludes the proof of Theorem 4.9 for the special case $Q = //z[b]/a$.

Remark (Regarding the proof of the general case). The proof of Theorem 4.9 for the general case, similarly to the proof of the special case given above, is also based on an MDS reduction from the delayed intersection problem. However, the document structure should be extended, similarly to the extension we made in the proof of Theorem 4.5 (see Appendix A) to ensure all other nodes in the query (except for z, a, b) can be matched. Therefore all subtrees surrounding the $//z/b$ segment in the query should be replicated in the document around each z-element.

4.4 Pattern matching

In this section we present two lower bounds for computing all the matches of basic twig queries. First, we show that computing all the matches of path queries that have at least one non-leaf descendant-axis node requires $\Omega(\text{docDepth})$ space. This lower bound is matched by the PathStack algorithm of Bruno *et al.* [6]. Then, we prove that the situation with tree queries is quite different: computing the matches of any tree query (regardless of the axis pattern) requires $\Omega(\text{outputSize})$ space.

4.5 Pattern matching for path queries

We now prove that computing all the matches of a path query that has at least one non-leaf descendant-axis node requires $\Omega(\text{docDepth})$ space. For the case the non-leaf descendant-axis node is followed by a child-axis node, the lower bound immediately follows from the $\Omega(\text{docDepth})$ space lower bound in the filtering mode (Theorem 4.5). What we prove here is that even if the axis pattern of the path is $(/)^*(//)(/)^+$ (i.e., the path ends with two or more descendant-axis nodes), then $\Omega(\text{docDepth})$ space is needed. For such queries (e.g., $//a//b$), we therefore have a gap between evaluation in the filtering mode, in which constant space is sufficient, and evaluation in the pattern matching mode, where $\Omega(\text{docDepth})$ space is needed. The lower bound is matched by the PathStack and TwigStack algorithms of Bruno *et al.* [6].

Remark. Actually the lower bound for queries with axis pattern $(/)^*(//)(/)^+$ is $\Omega(\min(\text{docDepth}, \text{outputSize}))$ (see Table 1). It means that there exist documents, for which docDepth $>>$ outputSize, that require at least outputSize space, and other documents, for which docDepth $<=$ outputSize, that require at least docDepth space. For clarity, we show here only the proof of the $\Omega(\text{docDepth})$ lower bound for the latter type of documents. The $\Omega(\text{outputSize})$ lower bound can be shown in a similar way.

To gain some intuition for the hardness of queries like $//a//b$, consider a document that contains a path of m a's followed by n b's. In order to output all the mn matches of the query $//a//b$ in this document, any algorithm will have either to store all the a's before starting to read the b's or vice versa. This gives a $\min\{m, n\}$ space lower bound. The intuition is formalized in the following theorem:

Theorem 4.14. *Let Q be any path query of the form $/c_1/c_2/\ldots/c_\ell//a_1//a_2\ldots//a_k$, where $\ell \geq 0$, $k \geq 2$, and $c_1, \ldots, c_\ell, a_1, \ldots, a_k$ are distinct labels. Then, for every algorithm for PM_Q and for every $d \geq 1$, there exists a document of depth at most d, on which the algorithm uses at least $\Omega(d)$ bits of space.*

We start with a proof of the theorem for the special case $Q = //a//b$. We then extend the proof to deal with arbitrary queries in the above XPath fragment. The proof is based on an MDS reduction from the tensor product problem:

Definition 4.15 (Tensor product)**.** Given two binary vectors x, y of lengths m and n, respectively, the *tensor product* of x and y, denoted $x \otimes y$, is a vector of length mn whose (i,j)-th entry is $x_i \cdot y_j$. The required

output is a list of indices of the set bits in $x \otimes y$, in arbitrary order. The tensor product of k vectors $x_1, \ldots, x_k$ is defined similarly by induction.

The following is a space lower bound for the tensor product problem in the MDS model. We assume x is given on one stream and y on another stream. The proof appears in Section 5.4.

Theorem 4.16. *The space complexity of computing $x \otimes y$ (x and y are of lengths m and n, respectively) in the* MDS *model is at least* $\min(m, n) - 3$ *bits.*

The reduction. We prove Theorem 4.14 for the case $Q = //a/b$ by an MDS reduction from the tensor product problem to PM_Q. Let $m = n = d/2$ (m, n are the lengths of the input to $x \otimes y$). The MDS reduction is based on the following functions:

- $r^1_{in}(x)$ and $r^2_{in}(y)$ construct the index streams T_a and T_b, respectively, of the XML document $E(x, y)$ (see Figure 4). $E(x, y)$ is the same for all (x, y), except for the labeling of elements. When $x_i = 1$, the corresponding element s_i is labeled 'a', and otherwise it is labeled 'c'. When $y_i = 1$, the corresponding element t_i is labeled 'b', and otherwise it is labeled 'd'.
- r_{out}: its input is a stream of matches, i.e., pairs of (a, b)-elements, from the document E, and the output is a list of indices that correspond to the position of the a and b elements in the input. Specifically, an input match (s_i, t_j) is translated on-the-fly to the output: "$(x \otimes y)_{i,j} = 1$". Therefore, $\text{DSS}(r_{out}) \leq \log n$.

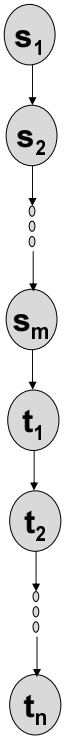

Figure 4: The document $E(x, y)$.

Claim 4.17. $\text{DSS}(r^1_{in}) = \text{DSS}(r^2_{in}) \leq \log n$.

Proof. We describe algorithms A and B for r^1_{in} and r^2_{in}, resp. In order to output the next tuple in T_a [T_b], A [B] advances the stream x [y] to the next set bit. If the position of this bit is i, the algorithm creates the tuple $(i, 2m + 2n + 1 - i, i)$ in T_a [$(m + j, m + 2n + j - i, m + j)$ in T_b]. If no set bit is found in x [y], the algorithm creates a T_a.End [T_b.End] tuple.

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the "Begin" attribute, and that they represent the document $E(x, y)$ whose depth is d. The space needed for A [B] is $\log n$ bits for keeping the current position in x [y]. □

Proposition 4.18. $(x \otimes y)_{i,j} = 1$ *iff the match* $(a \mapsto s_i, b \mapsto t_j)$ *belongs to* $\text{PM}_Q(E(x, y))$.

Proof. $(x \otimes y)_{i,j} = 1$ if and only if $x_i = y_j = 1$. This means that in $E(x, y)$ the label of s_i is 'a' and the label of t_j is 'b'. The latter happens if and only if $(a \mapsto s_i, b \mapsto t_j)$ is a match of *//a//b* in $E(x, y)$. □

Since r^1_{in} and r^2_{in} construct the index streams T_a and T_b of $E(x, y)$, it follows from the proposition above that:

$$r_{out}(\text{PM}_Q(r^1_{in}(x), r^2_{in}(y))) = r_{out}(\text{PM}_Q(E(x, y))) = x \otimes y$$

Therefore, by Corollary 4.4,

$$\text{MDSS}(\text{PM}_Q) \geq \text{MDSS}(x \otimes y) - \text{DSS}(r^1_{in}) - \text{DSS}(r^2_{in}) - \text{DSS}(r_{out}) \geq n - 3 - O(\log n) = \Omega(d)$$

We have proved Theorem 4.14 for the special case $Q =$ *//a//b*. The extension of the proof to arbitrary path queries of the above form is quite straightforward. We therefore provide only a proof sketch. The proof relies on a reduction from the tensor product of k vectors. For the latter, we have the following lower bound, which is proven in Section 5.4:

Theorem 4.19. *Given k vectors $x_1, x_2, \cdots, x_k$ of dimensions $m_1, \ldots, m_k$, respectively, the space complexity of computing $x_1 \otimes x_2 \otimes \cdots \otimes x_k$ in the* MDS *model is at least $\sum_{i=1}^{k} m_i - \max_i\{m_i\} - \log k - k + 1$.*

Proof of Theorem 4.14 (Sketch). We use a very similar MDS reduction from the tensor product of k vectors $x_1, \cdots, x_k$ of lengths $m_1, \cdots, m_k$, where $\sum_{i=1}^{k} m_i = d - \ell$. Similarly to the document $E(x, y)$ we constructed above (see Figure 4), the document $E(x_1, \ldots, x_k)$ we construct now is a path. There are two differences, however: (1) $E(x_1, \ldots, x_k)$ consists of k nested chains of elements (corresponding to $a_1, \ldots, a_k$) rather than just two; and (2) the path begins with the prefix $/c_1/c_2 \ldots /c_\ell$. □

4.6 Pattern matching for tree queries

As opposed to the filtering mode, pattern matching of all tree queries incurs high space costs. For a query Q and a document D, we define the *size* of $\text{PM}_Q(D)$ to be the number of distinct elements that occur in the matches in $\text{PM}_Q(D)$. We prove:

Theorem 4.20. *Let Q be any basic twig query, which is a tree (i.e., has at least two leaves). Then, for every algorithm for* PM_Q *and for every $s \geq 1$, there exists a document D, for which the size of* $\text{PM}_Q(D)$ *is at most s and on which the algorithm uses at least $\Omega(s)$ bits of space.*

This theorem too is proven via an MDS reduction from the tensor product problem. We first prove the theorem for the special case $Q =$ */a*[b and c].

The reduction. Let $n = m = \frac{s-1}{2}$ (m, n are the lengths of x, y, resp.). The MDS reduction is based on the following functions:

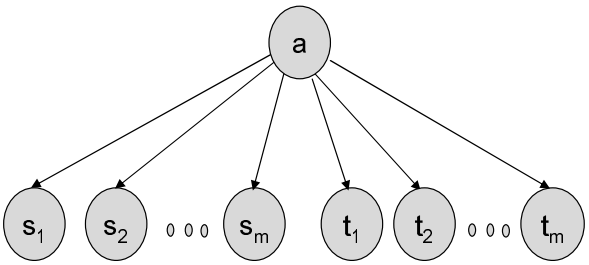

Figure 5: The document $F(x, y)$.

- $r^1_{in}(x)$ and $r^2_{in}(y)$ construct the index streams T_b and T_c, respectively, of the XML document $F(x, y)$ (see Figure 5). $F(x, y)$ is the same for all (x, y), except for the labeling of elements. When $x_i = 1$, the corresponding element s_i is labeled 'b', and otherwise it is labeled 'e'. When $y_i = 1$, the corresponding element t_i is labeled 'c', and otherwise it is labeled 'f'.
- r^1_c constructs the index stream T_a of the document $F(x, y)$, which consists of only one tuple. Note that T_a is fixed and does not depend on the input, and therefore $\text{DSS}(r^1_c) = O(\log n)$.
- r_{out}: its input is a stream of matches, i.e., tuples of (a, b, c)-elements, from the document E, and the output is a list of indices that correspond to the position of the b and c elements in the input. Specifically, an input match (a, s_i, t_j) is translated on-the-fly to the output: "$(x \otimes y)_{i,j} = 1$". Therefore, $\text{DSS}(r_{out}) = O(\log n)$.

Claim 4.21. $\text{DSS}(r^1_{in}) = \text{DSS}(r^2_{in}) \leq \log n$.

Proof. We describe algorithms A and B for r^1_{in} and r^2_{in}, resp. In order to output the next tuple in T_b [T_c], A [B] advances the stream x [y] to the next set bit. The corresponding tuple can be easily computed based on the position of that bit. If no set bit is found in x [y], the algorithm creates a T_b.End [T_c.End] tuple.

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the "Begin" attribute, and that they represent the document $F(x, y)$. Note also that the size of $\text{PM}_Q(F(x, y))$ is at most $2m+1 = s$, as required by the theorem. The space needed for A [B] is $O(\log n)$ bits for keeping the current position in x [y]. □

Proposition 4.22. $(x \otimes y)_{i,j} = 1$ *iff the match* $(a \mapsto a, b \mapsto s_i, c \mapsto t_j)$ *belongs to* $\text{PM}_Q(F(x, y))$.

Proof. $(x \otimes y)_{i,j} = 1$ if and only if $x_i = y_j = 1$. This means that in $F(x, y)$ the label of s_i is 'b' and the label of t_j is 'c'. The latter happens if and only if $(a \mapsto a, b \mapsto s_i, c \mapsto t_j)$ is a match of *//a//b* in $E(x, y)$. □

Since $r^1_c, r^1_{in}, r^2_{in}$ construct the index streams of $F(x, y)$, it follows from the proposition above that:

$$r_{out}(\text{PM}_Q(r^1_c(\epsilon), r^1_{in}(x), r^2_{in}(y))) = r_{out}(\text{PM}_Q(F(x, y))) = x \otimes y$$

Therefore, by Corollary 4.4,

$$\text{MDSS}(\text{PM}_Q) \geq \text{MDSS}(x \otimes y) - \text{DSS}(r^1_{in}) - \text{DSS}(r^2_{in}) - \text{DSS}(r^1_c) - \text{DSS}(r_{out}) \geq n - 3 - O(\log n) = \Omega(s)$$

In order to prove Theorem 4.20 for any tree query Q, we slightly change the reduction shown in the special case. Instead of the document $F(x, y)$, built on-the-fly (see Figure 5), now the document will be of the same structure and labels of Q, except for two query leaves, which will be replaced with the nodes $s_1, \ldots, s_m$ and $t_1, \ldots, t_m$ (as in the document $F(x, y)$).

5 The TMC model

In this section we present a new model of communication, the *token-based mesh communication model* (TMC), which can be used to prove space lower bounds in the MDS model. After investigating basic properties of protocols in the model, we use them to prove lower bounds for three problems: reverse set disjointness (Theorem 4.6), delayed intersection (Theorem 4.10), and tensor product (Theorems 4.16,4.19). We note that these lower bounds could have been proved directly through the MDS model, without using the new TMC model. However, we believe that the TMC model presents in an explicit way the various possible computations of a multiple data stream algorithm, i.e., the various cursor configurations, and enables cleaner and more modular proofs.

The TMC model. In the *token-based mesh communication* (TMC) model, there are n players, who wish to jointly compute a function f on a shared input $x \in \{0,1\}^m$. The players are placed on nodes of a network, whose underlying topology is a d-dimensional mesh. Specifically, the set of nodes is $V = [m_1] \times [m_2] \times \cdots \times [m_d]$, where $[m_i] = \{0, 1, \ldots, m_i\}$. Every node $(i_1, i_2, \ldots, i_d)$ has an outgoing edge to $(i_1, i_2, \ldots, i_d) + e_j$, for all j for which $i_j < m_j$. Here, e_j is the d-dimensional j-th standard unit vector.

The input $x \in \{0,1\}^m$ is viewed as a concatenation of d strings $x_1, x_2, \ldots, x_d$, where $x_i \in \{0,1\}^{m_i}$. Node $(i_1, i_2, \ldots, i_d)$ receives $(b_{1,i_1}, b_{2,i_2}, \ldots, b_{d,i_d})$, where $b_{j,k}$ is the k-th bit in x_j, for $1 \leq k \leq m_j$, and is 0, for $k = 0$.

The communication in the network is not in broadcast, as is in the more standard models, but is *token-based*. At each round of the protocol, a single player holds a "token", indicating she is the only one who can send messages in the round. She sends a single private message to one of her outgoing neighbors. The neighbor who receives the message holds the token at the next round. The communication always starts at the node $s = 0^d$ (the "start player") and ends at the node $t = (m_1, m_2, \ldots, m_d)$ (the "end player"). All players share a write-only output stream, to which only a player who holds the token can write. The stream should contain the value $f(x)$ by the end of the protocol.

The *max communication cost* of a protocol P in this model is the length of the longest message sent during execution of P on the worst-case choice of input x.

Figure 6 shows a 2-dimensional mesh, with $m_1 = m_2 = n$.

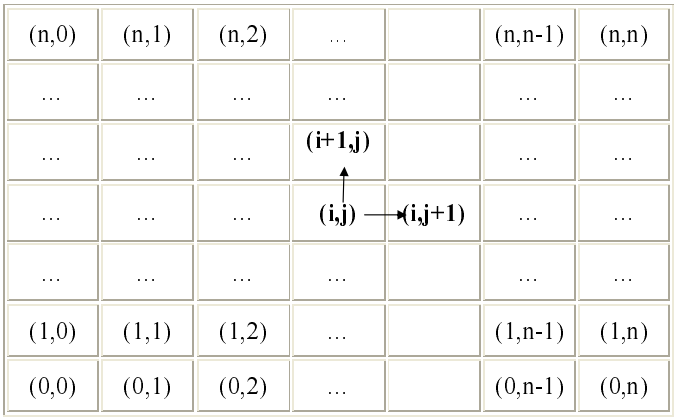

Figure 6: A 2-dimensional mesh, with $m_1 = m_2 = n$, in the TMC model.

The following shows a reduction from the TMC model to the MDS model:

Lemma 5.1 (Reduction lemma)**.** *Let $f : \{0,1\}^{m_1} \times \{0,1\}^{m_2} \times \cdots \times \{0,1\}^{m_d} \to B$. If there exists an algorithm that computes f in the* MDS *model with S bits of space, then there exists a protocol that computes*

f in the d-dimensional TMC *model whose max communication cost is at most S bits.*

Proof. Let A be an algorithm that computes f in the MDS model with S bits of space. The input of A is d streams $\{\mathbf{x_1}, \ldots, \mathbf{x_d}\}$ of sizes $\{m_1, \ldots, m_d\}$, respectively. In the TMC model we have a d-dimensional mesh, where every dimension corresponds to one stream. Each setting of the d cursors in the MDS model corresponds to one player in the d-dimensional mesh. We now describe a protocol P that computes f in the TMC model. At the beginning, player s (i.e., 0^d) holds the token, and it has received no input (by definition). It starts to execute the algorithm A. Whenever A moves a cursor, say the cursor of stream x_i, the player who currently holds the token, denoted as $(i_1, i_2, \ldots, i_d)$, sends the token together with the current content of the memory of A (S bits) to player $(i_1, i_2, \ldots, i_d) + e_i$. The player who receives the token and the memory-content continues the execution of A at the same way. Whenever A writes to the output stream, the simulating player does the same. Player t is the last to execute A, and it finishes the simulation. Note that P correctly computes f (because A does) and its max communication is S. □

5.1 Properties of the TMC model

We now investigate some basic properties of the TMC model, which are crucial to our lower bound proofs. For simplicity of exposition, we focus on 2-dimensional meshes, yet the definitions and results can be easily extended to d-dimensional meshes as well.

Let P be a protocol that computes $f(x, y)$ in the 2-dimensional TMC model.

Definition 5.2 (Communication path)**.** The *communication path* of protocol P on input (x, y), denoted $\text{PATH}(x, y)$, is the sequence of players $\{(i, j)\}$ through whom the token passes during the execution of P on (x, y).

The following fact states that all communication paths must pass through the diagonals $i + j = C$:

Proposition 5.3. *$\forall 1 \leq C \leq \min(m_1, m_2)$ and $\forall x, y$, $\exists\, 0 \leq i \leq C$, such that $(i, C - i) \in \text{PATH}(x, y)$.*

Proof. The diagonal $i + j = C$ is an (s, t)-cut, and thus any path from s to t crosses this cut. □

Definition 5.4 (Passing set)**.** The *passing-input-set* of protocol P w.r.t. player (i, j), denoted $\text{PASS}(i, j)$, is the set of all inputs (x, y) s.t. $(i, j) \in \text{PATH}(x, y)$.

Let $\text{PREF}_i(x)$ denote the first i bits of x, and let $\text{SUFF}_i(x)$ denote the last i bits of x, for $0 \leq i \leq m_1$. For $i = m_1 + 1$, $\text{SUFF}_{m_1+1}(x)$ is $0 \circ x$. (The same goes for m_2.)
Let player $(i, j) \in \text{PATH}(x, y)$. $\text{MSG}_{i,j}(x, y)$ denotes the message sent by player (i, j) during the execution of P on input (x, y). $\text{SUCC}_{i,j}(x, y)$ denotes the successor of (i, j) in $\text{PATH}(x, y)$.

Definition 5.5 (Packet)**.** Let player $(i, j) \in \text{PATH}(x, y)$. The *packet* sent by (i, j), denoted $\text{PACKET}_{i,j}(x, y)$, is defined as the combination of $\text{MSG}_{i,j}(x, y)$ and $\text{SUCC}_{i,j}(x, y)$.

Definition 5.6 (Prefix set)**.** The *passing-prefix-set* of protocol P w.r.t. player (i, j), denoted $\text{PREF}(i, j)$, is the set: $\{(\text{PREF}_i(x), \text{PREF}_j(y)) \mid (x, y) \in \text{PASS}(i, j)\}$

Proposition 5.7. $|\text{PASS}(i, j)| = |\text{PREF}(i, j)| \cdot 2^{(m_1 - i) + (m_2 - j)}$

Proof. For every $(\alpha, \beta) \in \text{PREF}(i, j)$, $\alpha \in \{0, 1\}^i$ and $\beta \in \{0, 1\}^j$, there are $2^{(m_1 - i) + (m_2 - j)}$ different continuations to inputs in $\text{PASS}(i, j)$. □

The following lemma states that the execution (i.e., packets and output) from any given node on the communication path depends solely on the message received by that node and on the input bits from that node and forward. Given that, the future execution is conditionally independent of the history, i.e., of previous messages and input bits.

Lemma 5.8 (History independence)**.** *Let* $(x, y), (x', y') \in \text{PASS}(i, j)$*. If* $\text{PACKET}_{i,j}(x, y) = \text{PACKET}_{i,j}(x', y')$*,* $\text{SUFF}_{m_1-i+1}(x) = \text{SUFF}_{m_1-i+1}(x')$*, and* $\text{SUFF}_{m_2-j+1}(y) = \text{SUFF}_{m_2-j+1}(y')$*, then the output written from the time player* (i, j) *sent his packet and until the end is the same on both inputs.*

Proof. Let $\text{PATH}_k(x, y)$ denote the k-th player in $\text{PATH}(x, y)$, for $0 \leq k \leq m_1 + m_2$. Note that, by the structure of the mesh, $(i, j) = \text{PATH}_{i+j}(x, y) = \text{PATH}_{i+j}(x', y')$. We now show that from the time player (i, j) sends his packet and until the end, the computation (i.e., packets) on both input pairs is exactly the same. It then follows that the output written during this time is also the same.

Specifically, we prove by induction on k, $k = i+j, \ldots, m_1+m_2$, that (i) $\text{PATH}_k(x, y) = \text{PATH}_k(x', y')$; and if (i', j') is the k-th player in both communication paths, then (ii) (i', j') sends the same packet on both input pairs.

For $k = i + j$, $\text{PATH}_{i+j}(x, y) = \text{PATH}_{i+j}(x', y') = (i, j)$, and we know that $\text{PACKET}_{i,j}(x, y) = \text{PACKET}_{i,j}(x', y')$. By the induction assumption, for any $k < (m_1 + m_2)$, $\text{PATH}_k(x, y) = \text{PATH}_k(x', y') = (i', j')$, where $i' \geq i$ and $j' \geq j$, and $\text{PACKET}_{i',j'}(x, y) = \text{PACKET}_{i',j'}(x', y')$. We prove for $k + 1$. (i) Since $\text{PACKET}_{i',j'}(x, y) = \text{PACKET}_{i',j'}(x', y')$, and it contains the destination node, then $\text{PATH}_{k+1}(x, y) = \text{PATH}_{k+1}(x', y')$. Let (p, q) be the player at position $k + 1$ in both communication paths. (ii) The packet that player (p, q) sends depends only on the message it has received from (i', j') (and it is the same in both cases), and on its input bits, which are x_p or x'_p, and y_q or y'_q. Since $p \geq i$ and $q \geq j$, and both inputs have the same $(m_1 - i + 1)$ and $(m_2 - j + 1)$ suffixes, then $x_p = x'_p$ and $y_q = y'_q$. Therefore player (p, q) sends the same packet.

□

5.2 Lower bound for Reverse set disjointness

We now use the TMC model to prove the space lower bound for reverse set disjointness in the MDS model.

Theorem 4.6 (restated) *For any* $n \geq 7$*, the space complexity of* RDISJ_n *in the* MDS *model is at least* $n - \log(n + 1) - 3$*.*

Proof. We will prove that any 2-dimensional TMC protocol computing the reverse set disjointness problem has a max communication cost of at least $m = n - \log(n + 1) - 3$ bits. Using Lemma 5.1, this would imply the same lower bound in the MDS model.

To reach a contradiction, we assume there exists a protocol P that solves RDISJ_n with max communication of m bits, where $m < n - \log(n + 1) - 3$. We will prove that there must be an input on which P errs.

According to Proposition 5.3, all communication paths go through the diagonal $i + j = n$. There are 2^{2n} different inputs (x, y), which means there are 2^{2n} communication paths, while there are $n+1$ players in this diagonal, i.e., players of the form $(i, n - i)$. Therefore, by the pigeonhole principle, there exists $0 \leq i \leq n$, s.t. $|\text{PASS}(i, n - i)| \geq \frac{2^{2n}}{n+1}$. We call this player $(i, n - i)$ *the congested player*.

Claim 5.9. *Let* $(i, n - i)$ *be the congested player of* P*. Then* $|\text{PREF}(i, n - i)| \geq \frac{2^n}{n+1}$

Proof. By Proposition 5.7, $|\text{PREF}(i, n - i)| = \frac{|\text{PASS}(i,n-i)|}{2^n} \geq \frac{2^{2n}}{2^n(n+1)}$ □

Let (x, y) be any input in $\text{PASS}(i, n - i)$. Consider the i-th bit of x and the $(n - i)$-th bit of y. There are four possible settings for these bits, inducing a partition of $\text{PREF}(i, n - i)$ into four sets. By the pigeonhole principle, one of these sets is of size at least $\frac{2^{n-2}}{n+1}$. Call this set A. Since the message sent by the congested player $(i, n - i)$ has at most m bits, where $m < n - \log(n + 1) - 3$, and since this player has only two neighbors, the number of possible packets it can send is less than $\frac{2 \cdot 2^{n-3}}{n+1}$. Since there are $\frac{2^{n-2}}{n+1}$ different

prefix pairs in A, then by the pigeonhole principle, there exist two pairs of prefixes $(\alpha', \beta'), (\alpha'', \beta'') \in A$ s.t. $\text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'')$, and $\alpha'_i = \alpha''_i$, and $\beta'_{n-i} = \beta''_{n-i}$.

The above prefix pairs (α', β') and (α'', β'') differ in at least one bit. W.l.o.g., we assume this is the k-th bit in α' and α'', where $k < i$. W.l.o.g., assume $\alpha'_k = 1$ and $\alpha''_k = 0$. We now define two different inputs for P, on one of which P must err. Let $\gamma = 0^{n-i}$ and $\delta = 0^{i-k} \circ 1 \circ 0^{k-1}$.

Claim 5.10. *P outputs the same answer on the two inputs: $(\alpha' \circ \gamma, \beta' \circ \delta)$ and $(\alpha'' \circ \gamma, \beta'' \circ \delta)$.*

Proof. First examine the execution (and possible output) until player $(i, n-i)$ sends its packet. Since the length of α' and α'' is i, and the length of β' and β'' is $n-i$, then there is no j s.t. both $(\alpha' \circ \gamma)_j$ and $(\beta' \circ \delta)^R_j$ belong to the prefix pair (α', β') (the same goes for α'' and β''). This implies that the value of RDISJ_n on these two input pairs is not necessarily 1. In addition, recall that $\alpha'_k = 1$, implying that the value of RDISJ_n on $(\alpha' \circ \gamma, \beta' \circ \delta)$ is not necessarily 0. Therefore, in order to determine $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta)$, one needs to know $(\beta' \circ \delta)^R_k = \delta^R_k$, which is unavailable yet. Hence the output bit on $(\alpha' \circ \gamma, \beta' \circ \delta)$ was not written yet.

Now recall that $\text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'')$, and that the $(n-i, i)$-suffix pairs (i.e., $(\alpha'_i \circ \gamma, \beta'_{n-i} \circ \delta)$) are the same for the two executions. Therefore, by Lemma 5.8, P outputs the same value[5]. □

Claim 5.11. $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta) \neq \text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta)$.

Proof. Since γ is all zero, and the only '1' bit in δ is in position $(i+1-k)$, which is the k-th bit in δ^R, then $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta) = 1$ iff α'_k is 1. Similarly, $\text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta) = 1$ iff α''_k is 1. Recall that α' and α'' differ in the k-th bit. Therefore, only one of $\text{RDISJ}_n(\alpha' \circ \gamma, \beta' \circ \delta)$ and $\text{RDISJ}_n(\alpha'' \circ \gamma, \beta'' \circ \delta)$ equals 1. □

Claims 5.10 and 5.11 prove that there is no protocol that solves RDISJ_n with less than $n - \log(n+1) - 3$ bits of space. This concludes the proof of Theorem 4.6. □

5.3 Lower bound for delayed intersection

Theorem 4.10 (restated) *For any $n \geq 7$, the space complexity of DINT_n in the MDS model is at least $n - \log(n+1) - 3$.*

Proof. We will prove that any 2-dimensional TMC protocol computing the delayed intersection problem has a max communication cost of at least $m = n - \log(n+1) - 3$ bits. Using Lemma 5.1, this would imply the same lower bound in the MDS model.

To reach a contradiction, we assume there exists a protocol P that solves DINT_n with max communication of m bits, where $m < n - \log(n+1) - 3$. We will prove that there must be an input on which P errs.

According to Proposition 5.3, all communication paths go through the diagonal $i + j = n$. There are 2^{3n+1} different inputs (s, t, u, v), which means there are 2^{3n+1} communication paths, while there are $n+1$ players in this diagonal, i.e., players of the form $(i, n-i)$. Therefore, by the pigeonhole principle, there exists $0 \leq i \leq n$, s.t. $|\text{PASS}(i, n-i)| \geq \frac{2^{3n+1}}{n+1}$. We call the player $(i, n-i)$ *the congested player*.

By Proposition 5.7, $|\text{PREF}(i, n-i)| = \frac{|\text{PASS}(i,n-i)|}{2^{2n+1}} \geq \frac{2^n}{n+1}$ (note that here $m_1 = 2n, m_2 = n+1$). Now consider the i-th bit of s and the $(n-i)$-th bit of u in any input in $\text{PREF}(i, n-i)$. There are four possible settings for these bits, inducing a partition of $\text{PREF}(i, n-i)$ into four sets. By the pigeonhole

[5] We claimed that the output bit was not written by the time player $(i, n-i)$ sends its packet only on $(\alpha' \circ \gamma, \beta' \circ \delta)$. Note that it possible that P on $(\alpha'' \circ \gamma, \beta'' \circ \delta)$ has already written the output bit before player $(i, n-i)$ sends its packet. However, since P on the two inputs writes the same output after player $(i, n-i)$ sends its packet, and it has to write the output bit for $(\alpha' \circ \gamma, \beta' \circ \delta)$, it means that P writes two output bits on $(\alpha'' \circ \gamma, \beta'' \circ \delta)$, and therefore obviously errs. Hence we can discard this case

principle, one of these sets is of size at least $\frac{2^{n-2}}{n+1}$. Call this set A. Since the message sent by the congested player $(i, n-i)$ has at most m bits, where $m < n - \log(n+1) - 3$, and since this player has only two neighbors, the number of possible packets it can send is less than $\frac{2 \cdot 2^{n-3}}{n+1}$. There are $\frac{2^{n-2}}{n+1}$ different prefix pairs in A, therefore by the pigeonhole principle, there exist two prefix pairs $(\alpha', \beta'), (\alpha'', \beta'') \in A$ s.t. $\text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'')$, and $\alpha'_i = \alpha''_i$, and $\beta'_{n-i} = \beta''_{n-i}$.

We now define two different inputs for P, on one of which P must err. The above prefix pairs (α', β') and (α'', β'') differ in at least one bit. There are two cases, based on the position of this bit:

- First assume this is the k-th bit in α' and α'', where $k < i$. W.l.o.g., assume $\alpha'_k = 1$. We will show that P did not output the k-th bit of DINT_n, denoted $\text{DINT}_n[k]$, on (α', β') by the time player $(i, n-i)$ sends its packet. For every input (s, t, u, v) s.t. $\text{PREF}_i(s \circ t) = \alpha'$ and $\text{PREF}_{n-i}(u \circ v) = \beta'$, $\text{DINT}_n[k](s, t, u, v) = (s_k \wedge v) = (\alpha'_k \wedge v) = v$. Note that v is the $n+1$-st bit in $(u \circ v)$ and the length of β' is $n - i \leq n$. Therefore, v is not a part of β', implying that after seeing (α', β'), no algorithm can determine $v = \text{DINT}_n[k](s, t, u, v)$.

  Let $\gamma \in \{0,1\}^{2n-i}$, $\delta \in \{0,1\}^i \times \{1\}$. We next show that P must output the same answer for $\text{DINT}_n[k]$ on the two inputs: $(\alpha' \circ \gamma, \beta' \circ \delta)$ and $(\alpha'' \circ \gamma, \beta'' \circ \delta)$. Recall that $\text{PACKET}_{i,n-i}(\alpha', \beta') = \text{PACKET}_{i,n-i}(\alpha'', \beta'')$, and that the suffixes $\alpha'_i \circ \gamma$ and $\beta'_{n-i} \circ \delta$ are the same for the two executions. Therefore, by Lemma 5.8, P outputs the same value.

  On the other hand, we show that $\text{DINT}_n[k](\alpha' \circ \gamma, \beta' \circ \delta) \neq \text{DINT}_n[k](\alpha'' \circ \gamma, \beta'' \circ \delta)$. This would imply that P errs on at least one of the inputs. Recall that $\text{DINT}_n[k](\alpha' \circ \gamma, \beta' \circ \delta) = (\alpha'_k \wedge v) = \alpha'_k$, and similarly $\text{DINT}_n[k](\alpha'' \circ \gamma, \beta'' \circ \delta) = \alpha''_k$. Since $\alpha'_k \neq \alpha''_k$, then the corresponding value of $\text{DINT}_n[k]$ is also different.

- The proof for the other case, where the two prefix pairs (α', β') and (α'', β'') differ in the k-th bit in β' and β'', is very similar. Either $\beta'_k = 1$ or $\beta''_k = 1$. For example, assume the former. The value of $\text{DINT}_n[n+k]$ on inputs whose prefix pair is (α', β') depends on the k-th bit of t, which is not available at the time the algorithm reads (α', β'). Hence, the protocol P can not output $\text{DINT}_n[n+k]$ on (α', β') by the time player $(i, n-i)$ sends its packet.

  We choose a suffix pair (γ, δ) for the two prefixes, such that the k-th bit in t, i.e., the $n - i + k$-th bit in γ, is set. $\text{DINT}_n[n+k]$ is different on the two inputs $(\alpha' \circ \gamma, \beta' \circ \delta)$ and $(\alpha'' \circ \gamma, \beta'' \circ \delta)$.

□

5.4 Lower bound for tensor product

We start with a proof of the lower bound for the tensor product of two vectors:

Theorem 4.16 (restated) *The space complexity of computing $x \otimes y$ (x and y are of lengths m and n, respectively) in the* MDS *model is at least* $\min(m, n) - 3$ *bits.*

Proof. We prove the lower bound in the TMC model. The corresponding lower bound in the MDS model would then follow from the reduction lemma (Lemma 5.1). To reach a contradiction, we assume there exists a protocol P that computes $x \otimes y$ with max communication of s bits, where $s < \min(m, n) - 3$. We shall prove that there exists an input (x, y) on which P must err.

We consider only inputs in which the last bit is set. The number of inputs is therefore 2^{m+n-2}. For every input (x, y), the communication path goes either through player $(m, n-1)$ or through $(m-1, n)$. This induces a partition of the inputs into two sets: $\text{PASS}(m, n-1)$, $\text{PASS}(m-1, n)$. By the pigeonhole principle, one of these sets is of size at least $\frac{2^{m+n-2}}{2}$. W.l.o.g., we assume it is $\text{PASS}(m, n-1)$. There are 2^n

possible values for y, which induces a partition of $\text{PASS}(m, n-1)$ into 2^n sets. By the pigeonhole principle, one of these sets is of size at least $\frac{2^{m+n-2}}{2 \cdot 2^n}$. Call this set A. Since the message sent by player $(m, n-1)$ has at most s bits, where $s < \min(m, n) - 3$, then by the pigeonhole principle, there exist two pairs of inputs $(x', y), (x", y) \in A$ s.t. $\text{PACKET}_{m,n-1}(x', y) = \text{PACKET}_{m,n-1}(x", y)$. x' and $x"$ differ in at least one bit, let it be the k-th bit, $k < m$.

Since y_n has not been read yet, the results of $(x \otimes y)_{*,n}$ could not have been written to the output stream yet. Recall that $x'_m = x''_m = 1$, $y_n = 1$, and $\text{PACKET}_{m,n-1}(x', y) = \text{PACKET}_{m,n-1}(x'', y)$. Therefore, player (m, n) writes the same output for the two inputs: (x', y) and (x'', y). But since $x'_k \neq x''_k$, then $(x' \otimes y)_{k,n} \neq (x'' \otimes y)_{k,n}$.

This proves that there is no algorithm that computes $x \otimes y$ with less than $\min(m, n) - 3$ bits of space. □

We now extend the proof to deal with $k \geq 2$ vectors:

Theorem 4.19 (restated) *Given k vectors $x_1, x_2, \cdots, x_k$ of dimensions $m_1, \ldots, m_k$, respectively, the space complexity of computing $x_1 \otimes x_2 \otimes \cdots \otimes x_k$ in the* MDS *model is at least $\sum_{i=1}^{k} m_i - \max_i\{m_i\} - \log k - k + 1$.*

Proof. We give a proof sketch, since it is an extension of the above proof for two vectors. We assume there exists a protocol P that computes $x_1 \otimes \cdots \otimes x_k$ with max communication of s bits, where $s < \sum_{i=2}^{k} m_i - \log k - k + 1$. We shall prove that there must be an input on which P errs.

We consider only input vectors in which the last bit is set ($2^{\sum_{i=1}^{k}(m_i - 1)}$ inputs). The last position of the communication path, before reaching player $(m_1, \cdots, m_k)$, induces a partition of the inputs into k sets. One of these sets is of size at least $\frac{2^{\sum_{i=1}^{k}(m_i-1)}}{k}$. W.l.o.g., we assume it is $\text{PASS}(m_1 - 1, m_2, \cdots, m_k)$. There are $2^{m_1 - 1}$ possible values for x_1, which induces another partition, in which one of the sets is of size at least $\frac{2^{\sum_{i=2}^{k}(m_i-1)}}{k}$. Call this set A. By the pigeonhole principle, there exist two inputs $I_1 = (x_1, x'_2, \cdots, x'_k)$ and $I_2 = (x_1, x"_2, \cdots, x"_k)$ in A, s.t. the packet sent by player $(m_1 - 1, m_2, \cdots, m_k)$ is the same on these inputs. Therefore, the last player writes the same output for the two inputs, but since I_1 and I_2 differ in at least one bit, then their tensor products also differ in the corresponding entry. □

6 Algorithms

In this section we present our upper bounds for evaluating twig queries, for the three evaluation modes.

6.1 The filtering algorithm

We now present a constant space filtering algorithm for queries that do not contain the (//)(/) pattern. The algorithm, described in Figure 7, computes $\text{FILTER}_Q(D)$ by trying to find a match of Q in D. The basic procedure used in the algorithm is NextMatchUnderSelf(u, e_u), which gets as input a query node u and the element e_u, on which the cursor of the stream T_u is currently positioned[6], and returns true if and only if the sub-query Q_u has a match in the sub-document D_{e_u}. Moreover, if such a match exists, the procedure advances the stream cursors to the positions that indicate the match.

NextMatchUnderSelf works by recursively searching for matches of the sub-queries rooted at the children of v. To this end, it calls the procedure NextMatchUnderParent(v, e_u). The latter gets as input a query node v and the element e_u, on which the cursor of T_u ($u = \text{parent}(v)$) is currently positioned, and returns true if and only if Q_v has a match in D_{e_v}, where e_v is a descendant of e_u whose relationship with e_u matches

[6] As mentioned in Section 3, we use T_u to denote the cursor on the stream corresponding to u. This way, if two query nodes u, v share the same label, then T_u and T_v denote two separate cursors on the same stream.

the axis of v. This procedure works by repeatedly advancing the cursor of T_v, until finding the desired element e_v. If a match is found, the cursors of the corresponding streams are advanced to positions that indicate the match.

```
1:Function Filter(Q, D)
2:   return NextMatchUnderSelf(root(Q), root(D))

1:Function NextMatchUnderSelf(u, e_u)
2:   for every child v of u
3:     if ( !NextMatchUnderParent(v, e_u) )
4:       return false
5:   return true

1:Function NextMatchUnderParent(v, e_u)
2:   e_v := T_v.ReadElement()
3:   while ((e_v != T_v.End) and (e_v.Begin < e_u.End))
4:     if ( (relationship between e_v and e_u matches
           axis(v)) and NextMatchUnderSelf(v, e_v) )
5:       return true
6:     T_v.Advance()
7:     e_v := T_v.ReadElement()
8:   return false
```

Figure 7: Filtering algorithm for queries without (//)(/).

The axis pattern of Q allows the algorithm to decide locally whether an element participates in a match or not, without having to remember elements for later use. The intuition is that a child-axis node can be matched only to elements of the same level, since all of its query-ancestors are also child-axis nodes. Therefore recursions in the document, which are the cause of difficulty in the pattern *//a/b*, are irrelevant for this case.

6.1.1 Example run

In order to illustrate how the filtering algorithm works, we provide an example run. Consider the document and the query presented in Figure 8. A subscript is added to each element to indicate the order in the index stream. Initially, the three cursors point to (a_1, b_1, c_1). $NextMatchUnderSelf$ calls $NextMatchUnderParent(a, root(D))$, which searches for an a element that has b and c descendants. a_1 is the first element checked. Now T_b and T_c are advanced separately, until the cursors point to b and c elements that are descendants of a_1, or begin after a_1 ends. Since b_1 is not nested within a_1, T_b is advanced to b_2, which matches the required axis. However, c_1 begins after a_1 ends, and therefore a_1 is rejected as a possible match to a, and T_a is advanced to a_2. Now the three cursors point to (a_2, b_2, c_1). Again, we look for b and c descendants of a_2. b_2 is not nested within a_2, and T_b is advanced to b_3, which matches the required axis. c_1 is a already a descendant of a_2. Therefore both calls to $NextMatchUnderParent(b, a_2)$ and $NextMatchUnderParent(c, a_2)$ return true, which means that $NextMatchUnderParent(a, root(D))$ returns true. $Filter(Q)$ returns true and the three cursors point to (a_2, b_3, c_1), which indicate the match found.

6.1.2 Complexity analysis

Space complexity:

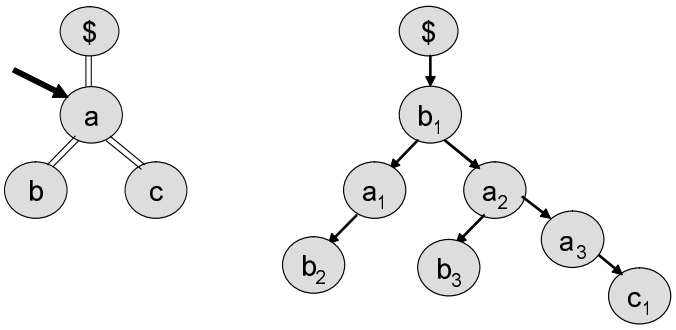

Figure 8: An example XML document (right) and query (left).

Proposition 6.1. *Let Q be a basic twig query that does not contain the (//)(/) pattern. Then, the space complexity of the algorithm $Filter(Q, D)$ is $\tilde{O}(1)$.*

Proof. The recursion depth of the algorithm is equal to the query depth. Each level requires space for storing $O(1)$ document elements. Therefore, the space complexity is $\tilde{O}(1)$. □

Time and I/O complexity: The time and I/O complexity of the algorithm are both linear in the length of the input streams. Note that the I/O complexity may be substantially decreased by using B+ trees over the index streams, and "$ForwardTo(pos)$" methods in the $NextMatchUnderParent$ method, similarly to previous algorithms (e.g., see [19]). However, since the focus of this research is the memory requirements, and it is quite simple to integrate these I/O saving methods, we present here the simplified version.

6.1.3 Correctness analysis

We now provide a full analysis of the algorithm's correctness:

Theorem 6.2. *Let Q be any basic twig query, which does not contain the (//)(/) pattern. Then, the algorithm Filter(Q,D) returns true if and only if there exists at least one match of Q in D.*

In order to prove Theorem 6.2, we need to show that the algorithm is both *sound* (returns true, only if a match exists) and *complete* (if a match exists, returns true).

Cursor configurations.

A notion that will play a crucial role in our analysis is *cursor configurations*. We denote by Q and D any basic twig query and any document, respectively. Let u be a query node. A *Q_u-cursor configuration* (or *Q_u-configuration*, in short) is a setting of the cursors $\{T_v\}_{v \in Q_u}$[7]. For a Q_u-configuration C and for a node $v \in Q_u$, $C[v]$ denotes the position of the cursor T_v as specified by C. We sometimes abuse notation and think of $C[v]$ as the document element pointed by this cursor.

A Q_u-configuration can be viewed as a mapping from Q_u to elements of D that preserves label matches. If this mapping is a match, we say that the configuration *induces a match*.

Let C_1, C_2 be two Q_u-configurations. C_1 is said to *dominate* C_2, denoted $C_1 \succeq C_2$, if for every $v \in Q_u$, $C_1[v] \geq C_2[v]$. As stream cursors move only in the forward direction, subsequent configurations encountered during an execution of an algorithm always dominate one another.

[7] We slightly abuse notation, and use T_v to denote both the stream $T_{\mathrm{label}(v)}$ and the cursor on this stream corresponding to v.

The Q_u-configuration at the time a function f is called is the *starting Q_u-configuration of f*. The Q_u-configuration when f returns is called the *ending Q_u-configuration of f*. By the above, the ending configuration always dominates the starting configuration.

We next analyze the main subroutine, NextMatchUnderSelf, and show it is sound and complete. It is easy to verify that whenever NextMatchUnderSelf(u, e_u) is called, then e_u is the element on which the cursor T_u is currently positioned.

Soundness.

The soundness of Filter(Q,D) will follow from the soundness of the function NextMatchUnderSelf:

Lemma 6.3 (Soundness)**.** *If NextMatchUnderSelf(u, e_u) returns true, then its ending Q_u-configuration induces a match of Q_u in D_{e_u}.*

Proof. We prove the lemma by induction on $k = \mathrm{height}(u)$. The base case, $k = 0$, corresponds to a leaf u. In this case the function always returns true, and indeed the mapping $u \mapsto e_u$ is a trivial match of Q_u in D_{e_u}. The ending Q_u-configuration equals in this case the starting Q_u-configuration. The latter induces the trivial match, by assumption.

Assume that the lemma holds for all nodes of height at most k. Let u be a node of height $k + 1$. Since the function returns true, then also the calls to NextMatchUnderParent(v, e_u), for every child v of u, return true. Consider one such child v.

Since NextMatchUnderParent(v, e_u) returns true, the function must have found an element e_v that satisfies the following: (1) the structural relationship between e_v and e_u matches $\mathrm{axis}(v)$; and (2) the function NextMatchUnderSelf(v, e_v) returns true. By the induction hypothesis, the latter implies that there is a match ϕ_v of Q_v in D_{e_v} and that the ending Q_v-configuration of the call to NextMatchUnderSelf(v, e_v) induces this match.

We can now define a match ϕ_u of Q_u in D_{e_u} as follows: (1) $\phi_u(u) = e_u$; (2) for every child v of u and for every node $w \in Q_v$, $\phi_u(w) = \phi_v(w)$. As each node in Q has its own separate cursor, then the ending Q_u-configuration of NextMatchUnderSelf(u, e_u) consists of the ending Q_v-configurations of NextMatchUnderParent(v, e_u), for each child v of u. The latter induce the matchings $\{\phi_v\}_v$ is a child of u and therefore the ending Q_u-configuration induces the matching ϕ_u. □

Completeness.

The completeness of Filter(Q,D) follows from the following lemma:

Lemma 6.4 (Completeness)**.** *Suppose Q does not contain the (//)(/) pattern and let C be the starting Q_u-configuration of NextMatchUnderSelf(u, e_u). If there exists a Q_u-configuration $C' \succeq C$ that induces a match of Q_u in D_{e_u}, then NextMatchUnderSelf(u, e_u) returns true.*

We assume from now on that Q does not contain the (//)(/) pattern. The following propositions are a key to proving the completeness lemma:

Proposition 6.5. *Let ϕ be a match of Q in D. Then, for every child-axis node u, $\mathrm{depth}(\phi(u)) = \mathrm{depth}(u)$.*

The proof is straightforward by induction on $\mathrm{depth}(u)$.

Proposition 6.6. *If during the execution of Filter(Q,D), NextMatchUnderSelf(u, e_u) is called with a child-axis node u, then $\mathrm{depth}(e_u) = \mathrm{depth}(u)$.*

Proof. We prove by induction on $k = \mathrm{depth}(u)$. The base case, $k = 0$, corresponds to the query root. In this case the function NextMatchUnderSelf(u,*) is only called with the document root as its second parameter, and indeed its depth is 0.

Assume that the lemma holds for all nodes of depth at most k. Let u be a node of depth $k+1$. The function NextMatchUnderSelf(u,e_u) can only be called from NextMatchUnderParent(u, e_v), where $e_v \in T_v$ and $v = \text{parent}(u)$ in Q, and only if e_u is a child of e_v. u is a child-axis node, therefore all of its ancestors are too (by the definition of Q). Since NextMatchUnderParent(u, e_v) can only be called from NextMatchUnderSelf(v, e_v), then by the induction hypothesis, $\text{depth}(e_v) = \text{depth}(v) = k$, which proves that $\text{depth}(e_u) = k+1$ (because e_u is a child of e_v).

□

Proof of Lemma 6.4. We prove the lemma by induction on $k = \text{height}(u)$. For $k = 0$, u is a leaf. In this case the function always returns true.

Suppose that the lemma holds for all nodes of height at most k. Consider a node u of height $k+1$. NextMatchUnderSelf(u, e_u) returns true only if the calls to NextMatchUnderParent(v, e_u), for each child v of u, return true. Consider such a child v, then.

Let C_v be the restriction of C to Q_v. Note that C_v is the starting Q_v-configuration of NextMatchUnderParent(v, e_u), even if v is not the first child to be processed. This is because a call to NextMatchUnderParent(v', e_u), for any other child v' of u, cannot change cursors corresponding to nodes in Q_v.

Let C'_v be the restriction of C' to Q_v. C'_v induces a match of Q_v in $D_{e'_v}$, where $e'_v = C'[v]$. Note that $e'_v \in D_{e_u}$ and its structural relationship with e_u matches $\text{axis}(v)$.

Since C' dominates C, then also C'_v dominates C_v. It follows that $C_v[v]$ precedes (or equals) e'_v in the stream T_v. When calling NextMatchUnderParent(v, e_u), the function enumerates the elements e_v on the stream T_v, starting with $C_v[v]$. We next show that the enumeration has to stop either at e'_v or before in success.

If the enumeration stops at some e_v that precedes e'_v, then NextMatchUnderParent(v, e_u) returns true. So suppose the enumeration has not stopped at any of these nodes. We would like to show it must stop at e'_v.

Claim 6.7. *Let C''_v be the Q_v-configuration when the algorithm starts processing e'_v, i.e., when NextMatchUnderParent(v, e_u) reaches line 3 and the cursor T_v points to e'_v. Then, $C''_v \preceq C'_v$.*

Before we prove this claim, let us use it to conclude the proof of Lemma 6.4. Since $C''_v[v] = e'_v$ and C''_v is dominated by C'_v, which induces a match of Q_v, then by the induction hypothesis, the function NextMatchUnderSelf(v, e'_v) returns true, and so does its calling function NextMatchUnderParent(v, e_u). We conclude that NextMatchUnderParent(v, e_u) returns true for all children v of u, and thus also NextMatchUnderSelf(u, e_u) returns true. □

Proof of Claim 6.7. Suppose, to reach a contradiction, that C''_v is not dominated by C'_v. This implies that there exists a node $b \in Q_v$ s.t. $C''_v[b] > C'_v[b]$. If there is more than one such node, we choose b to be the node, for which T_b is the first to be advanced beyond $C'_v[b]$. Let a be the parent of b in Q. T_b must be advanced beyond $C'_v[b]$ during the execution of NextMatchUnderParent(b, e''_a), where e''_a is some node in the stream T_a. Note that at the time T_b is advanced beyond $C'_v[b]$, the cursor T_a points to e''_a. By the choice of b, the position of e''_a in the stream T_a is at most $C'_v[a]$, i.e., e''_a.Begin $\leq C'_v[a]$.Begin.

There are two possible positions for e''_a in the document: (1) e''_a.End $< C'_v[a]$.Begin, or (2) e''_a is an ancestor of (or equals) $C'_v[a]$. We prove that both lead to a contradiction.

Consider the first option, i.e., e''_a.End $< C'_v[a]$.Begin. $C'_v[b]$ is nested within $C'_v[a]$ since C' induces a match. Therefore, e''_a.End $< C'_v[b]$.Begin, which means that the condition of the while loop in NextMatchUnderParent(b, e''_a) is not satisfied, and the function could not advance T_b beyond $C'_v[b]$, in contradiction to our assumption.

Consider then the second option, i.e., e''_a is an ancestor of (or equals) $C'_v[a]$. There are two sub-cases here. (i) a is a child-axis node and $e''_a \neq C'_v[a]$, (ii) a is a descendant-axis node, or (iii) $e''_a = C'_v[a]$. In case (i), e''_a is an ancestor of $C'_v[a]$ and therefore $\text{depth}(e''_a) < \text{depth}(C'_v[a])$. Since C' induces a match,

then according to Proposition 6.5, $\mathrm{depth}(C'_v[a]) = \mathrm{depth}(a)$, and thus $\mathrm{depth}(e''_a) < \mathrm{depth}(a)$. Therefore, based on Proposition 6.6, there is no call to NextMatchUnderSelf(a,e''_a), which means there is no call to NextMatchUnderParent(b,e''_a), in contradiction to the assumption.

To deal with cases (ii) and (iii), we first show that in both of them the relationship between $C'_v[b]$ and e''_a matches $\mathrm{axis}(b)$. In case (ii), a is a descendant-axis node. Hence, also b must be a descendant-axis node (Q does not have the (//)(/) pattern), and since C' induces a match, then $C'_v[a]$ is an ancestor of $C'_v[b]$. In addition, recall that e''_a is an ancestor of (or equals) $C'_v[a]$. Therefore, $C'_v[b]$ is a descendant of e''_a, and thus the relationship between $C'_v[b]$ and e''_a matches $\mathrm{axis}(b)$. In case (iii), $e''_a = C'_v[a]$, and thus the relationship between $C'_v[b]$ and e''_a matches $\mathrm{axis}(b)$, because C' induces a match.

Now consider the Q_b-configuration when NextMatchUnderParent(b, e''_a) starts processing $C'_v[b]$, i.e., when it reaches line 3 and the cursor T_b points to $C'_v[b]$. The relationship condition in line 4 is satisfied, therefore the function calls NextMatchUnderSelf(b,$C'_v[b]$). Since we assumed b is the node in Q_v whose cursor is the first to move beyond $C'_v[b]$, then the current Q_b-configuration is dominated by C'_b (the restriction of C' to Q_b). By the induction hypothesis, NextMatchUnderSelf(b,$C'_v[b]$) returns true. NextMatchUnderParent(b,e''_a) would also return true, without advancing T_b, in contradiction to our assumption. □

6.2 The full-fledged evaluation algorithm

In this section we extend the filtering algorithm we presented in Section 6.1, which evaluates queries that do not contain the (//)(/) pattern, into a full-fledged evaluation algorithm. The algorithm, presented in Figure 9, makes use of the NextMatchUnderSelf procedure from the filtering algorithm. The basic procedure of the algorithm is Eval(Q, t, D), which gets as input a query tree Q, its output node t, and a document D, and works by iteratively looking for a match of Q in D. For each match found, it: (i) outputs the document element e_t that t is mapped to by this match, and (ii) advances the cursor beyond e_t.

```
1:Function Eval(Q, t, D)
2:   while ( NextMatchUnderSelf(root(Q), root(D)) ) then
3:     output T_t.ReadElement()
4:     T_t.Advance()
```

Figure 9: FFE algorithm for queries without (//)(/).

6.2.1 Complexity analysis

Space complexity:

Proposition 6.8. *Let Q be a basic twig query that does not contain the (//)(/) pattern. Then, the space complexity of the algorithm $Eval$ is $\tilde{O}(1)$.*

Proof. The function $Eval$ is iteratively calling $NextMatchUnderSelf$, which uses constant space (see Proposition 6.1). Therefore, the space complexity of $Eval$ is $\tilde{O}(1)$. □

Time and I/O complexity: The time and I/O complexity of the algorithm are both linear in the length of the input streams. As we noted in Section 6.1.2, the I/O complexity may be easily decreased.

6.2.2 Correctness analysis

In order to prove the correctness of the algorithm Eval, we need to show it is *sound* (every element it outputs indeed matches t) and *complete* (every element that matches t node is output).

Soundness.

To prove soundness, let e_t be an element that Eval outputs. e_t must have been the element pointed by the cursor T_t after the function NextMatchUnderSelf returned true. By Lemma 6.3, the ending configuration of NextMatchUnderSelf (if it returns true) induces a match ϕ of Q in D. Therefore, $e_t = \phi(t)$ indeed matches t.

Completeness.

Let $e_{t_1}, e_{t_2}, ..., e_{t_k}$ be the elements that match t, in document order. We prove that Eval outputs them in this order.

Proof sketch. We show that the i-th call to NextMatchUnderSelf in line 2 of Eval advances the cursor configuration to the "minimum" match ϕ, for which $\phi(t)$.Begin $\geq e_{t_i}$.Begin. Here, the "minimum" is w.r.t. the partial order induced by the domination relation, and the existence of the minimum is guaranteed by the fact Q does not have the (//)(/) pattern. Since we always move to the minimum match, we are guaranteed not to miss a match of t with one of the e_{t_i}'s.

We now examine the execution of Eval(Q,t,D), which consists of a sequence of calls to NextMatchUnderSelf. Let C_i^s and C_i^e denote the starting and ending configuration of the i-th call, respectively.

Proposition 6.9. $\forall v \in Q$: *if* $v \neq o$ *then* $C_{i+1}^s[v] = C_i^e[v]$, *and for* o: $C_{i+1}^s[o] = C_i^e[o] + 1$.

Since every match m induces a configuration, we sometimes abuse notation and think of m as the induced configuration. Let M_i denote the set of all matches m of Q in D, for which $m[t] \geq e_{t_i}$. Recall that domination induces a partial order over the space of configurations [matches]. Therefore, not every set of configurations necessarily has a minimum (i.e., a configuration that is dominated by all other configurations in the set). The following lemma describes a scenario, where such a minimum always exists. The lemma will be used at the core of the completeness arguments below:

Lemma 6.10. *Suppose Q is a basic twig query, which does not contain the (//)(/) pattern. Then* $\forall 1 \leq i \leq k$, M_i *has a minimum, denoted* m_i.

Proof. In order to show M_i has a minimum, it suffices to prove that M_i has only one match that does not dominate any other match in M_i. Suppose, to reach a contradiction, M_i has two such matches, m_1 and m_2. We use m_1 and m_2 to define a new match m_3. For each $v \in Q$, $m_3[v] = \min\{m_1[v], m_2[v]\}$. Since $m_1, m_2 \in M_i$, i.e., $m_1[t], m_2[t] \geq e_{t_i}$, then also $m_3 \in M_i$. We would like to show that m_3 induces a match.

For each $v \in Q$, let m_3^v denote the restriction of m_3 to Q_v. We prove by induction on $\mathrm{height}(v)$, that for every such v, m_3^v induces a match. The base case, $\mathrm{height}(v) = 0$, corresponds to leaves. In this case, m_3^v consists of the single element $m_3[v]$, which by definition belongs to T_v, and thus matches v.

Assume, then, that for every node w of height k, m_3^w induces of a match of Q_w. Let v be a node of height $k+1$. By the induction hypothesis, for every child w of v, m_3^w induces a match of Q_w. In order for m_3^v to induce a match, we need to make sure that the structural relationship between $m_3[w]$ and $m_3[v]$ matches $\mathrm{axis}(w)$, for every child w of v.

Fix one such child w. If $m_3[w] = m_1[w]$ and $m_3[v] = m_1[v]$, then since m_1 induces a match, the structural relationship between $m_3[w]$ and $m_3[v]$ matches $\mathrm{axis}(w)$. The same thing happens if $m_3[w] = m_2[w]$ and $m_3[v] = m_2[v]$, due to the fact m_2 induces a match. Assume, then, that $m_3[w] = m_1[w] < m_2[w]$ and $m_3[v] = m_2[v] < m_1[v]$ (the opposite case is handled analogously). We therefore have the following situation: $m_2[v]$ precedes $m_1[v]$ in pre-order traversal. However, the child or descendant $m_2[w]$ of $m_2[v]$ *succeeds* the child or descendant $m_1[w]$ of $m_1[v]$. This can happen only if $m_1[v]$ is a descendant of $m_2[v]$. We now split to two cases, based on the axis of w.

- **Case 1:** $\text{axis}(w) = //$. In this case $m_1[w]$ is a descendant of $m_1[v]$ (since m_1 induces a match), which in turn is a descendant of $m_2[v]$ (by the above). Therefore, $m_3[w]$ is a descendant of $m_3[v]$, as desired.
- **Case 2:** $\text{axis}(w) = /$. In this case we have an element and its descendant, namely $m_2[v]$ and $m_1[v]$, both of which match the same query node v. This can happen only if v or one of its ancestors has a descendant axis. However, since w has a child axis and is a child of v, neither v nor its ancestors are allowed to have a descendant axis (recall that in our query no (//) is followed by a (/)). Therefore, this case simply cannot happen.

We conclude that $m_3 \in M_i$ induces a match. By definition, both m_1 and m_2 dominate m_3, in contradiction to our assumption that neither of them dominates any other configuration in M_i. The lemma follows. □

Lemma 6.11. $m_i[t] = e_{t_i}$.

Proof. Recall that for each $m \in M_i$, $m[t] \geq e_{t_i}$. In addition, $e_{t_i} \in \text{FFE}_Q(D)$, therefore $\exists m \in M_i$, s.t. $m[t] = e_{t_i}$. Since m_i is the minimum, then $m_i[t] = e_{t_i}$. □

Lemma 6.12. *Let $1 \leq i \leq k$. If $C_i^s[t] > e_{t_{i-1}}$ and $M_i \succeq C_i^s$, then $C_i^e = m_i$.*

Proof. Let us examine the starting configuration of the i-th call to NextMatchUnderSelf, i.e., C_i^s, and the set S of matches dominating it. By the lemma, $M_i \succeq C_i^s$. In addition, $C_i^s[t] > e_{t_{i-1}}$, therefore for any match m dominating C_i^s, $m \in M_i$. Thus $S = M_i$. The next property of NextMatchUnderSelf shows that its ending configuration is exactly m_i. □

Claim 6.13. *Suppose that we call NextMatchUnderSelf(u, e_u) with a starting Q_u-configuration C with $C[u] = e_u$. Let M_C denote the set of Q_u-configurations that: (1) dominate C; (2) induce a match of Q_u in D_{e_u}. If M_C has a minimum, then NextMatchUnderSelf(u, e_u) returns true and its ending Q_u-configuration is the minimum of M_C.*

Proof. By the completeness property of NextMatchUnderSelf (Lemma 6.4), we know that if M_C is not empty, then the function returns true. By Lemma 6.3, the ending configuration induces a match. We only need to prove that this match m is the minimum of M_C. Suppose, to reach a contradiction, that there is a different match $m' \in M_C$, which is the minimum. Now assume the following setting: we "end" all the streams right after the positions of m', and call NextMatchUnderSelf with the same starting configuration C. In this case $M_C = \{m'\}$, and by the correctness of NextMatchUnderSelf, the ending configuration is m'. Since the algorithm reads the streams sequentially, then until reaching configuration m', it can not differentiate between the two settings, and therefore in our original setting the ending configuration would also be m' and not m. □

Lemma 6.14. *For each $i \in [k]$: (*) $C_i^s[t] > e_{t_{i-1}}$ and (**) $M_i \succeq C_i^s$.*

Proof. We prove by induction on i. The base case corresponds to the first call to NextMatchUnderSelf, when the starting configuration is the beginning of all streams. In this case both (*) and (**) trivially hold. Assume the lemma holds for every $i \geq j$. By the induction hypothesis, $C_j^s[t] > e_{t_{j-1}}$ and $M_j \succeq C_j^s$. By Lemma 6.12, $C_j^e = m_j$. Note that before the $(j+1)$-th call only T_t is advanced, and $m_j[t] = e_{t_j}$ (Lemma 6.11), therefore $C_{j+1}^s[t] > e_{t_j}$ (*). $m_j \preceq M_{j+1}$, since m_j is the minimum of M_j and $M_{j+1} \subseteq M_j$. Note that m_j is identical to C_{j+1}^s except for the position of T_t, which was advanced only by one element. It follows that $C_{j+1}^s \preceq M_{j+1}$ (**). □

Corollary 6.15. *For each $i \in [k]$: $C_i^e = m_i$.*

The above corollary and Lemma 6.11 prove that for each $i \in [k]$: $C_i^e[t] = e_{t_i}$, which means that the i-th call to NextMatchUnderSelf outputs e_{t_i}. Thus the FFE algorithm outputs all $e_{t_i} \in \text{FFE}_Q(D)$, and this concludes the proof of completeness.

6.3 Pattern matching upper bound

In this section we present an upper bound for computing all the matches of basic twig queries.

We observe that existing algorithms ($TwigStack$ [6] and $TwigStackList$ [22]) use in some cases much more space than indicated by the output size lower bound. When the query contains only descendant axes, both algorithms keep only elements that are guaranteed to be in at least one match. However, when child axis nodes are involved, the algorithms may keep many redundant intermediate results. To demonstrate this sub-optimality, consider the query $Q =$ /a[b and c], and the document D depicted in Figure 10. There is no match for Q in D (i.e., the output size is 0), but both $TwigStack$ and $TwigStackList$ keep the n paths $(a_1, b_1), \ldots, (a_1, b_n)$ in memory until they reach their second phase, in which they merge path solutions.

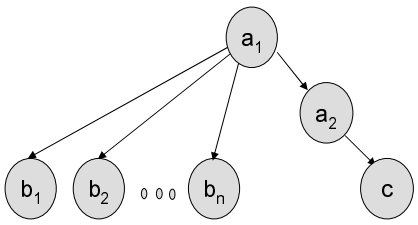

Figure 10: A document demonstrating the sub-optimality of $TwigStack$ and $TwigStackList$ on the query /a[b and c].

We now describe a minor modification to the known $TwigStack$ algorithm. The new version uses O(output- Size) space in the worst-case, for queries which do not consist of the (//)(/) pattern. Finding matching upper bounds for queries that contain the (//)(/) pattern remains an open problem.

We suggest the following modification. Note that each child axis node in this fragment can match only elements of the same depth (see Proposition 6.5). Therefore, we can filter its stream to read only such candidate elements. Done that, we can run twigstack as if all nodes have a descendant-axis. Now consider the example document and query presented in Figure 10. The suggested modification will result in automatically ignoring the element c, since its depth is not 2 (as in the query), and therefore $twigStack$ will not consider any of the b_i elements as candidates for a match, and therefore will not store them.

7 Conclusions

In this paper we initiated a systematic study of memory lower bounds for evaluating twig queries over indexed documents. We provide an analytical explanation for the difficulty in handling queries with child-axis nodes, and also point out the overhead incurred by algorithms that work in the pattern matching mode. We present a new algorithm that avoids this overhead, and achieves dramatic improvements in space for certain types of queries. We introduce a new model of communication complexity, the TMC model, through which we can prove space lower bounds for multiple data streams algorithms.

References

[1] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the Symposium on Foundations of Computer Science (FOCS)*, pages 540–549, 2004.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. Database Syst.*, 29(1):162–194, 2004.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the Symposium on Principles of Database Systems (PODS)*, pages 1–16, 2002.

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

[5] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the memory requirements of XPath evaluation over XML streams. *J. Comput. Syst. Sci.*, 73(3):391–441, 2007.

[6] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: optimal XML pattern matching. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 310–321, 2002.

[7] T. Chen, J. Lu, and T. W. Ling. On boosting holism in XML twig pattern matching using structural indexing techniques. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 455–466, 2005.

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

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

[10] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang. Optimizing cursor movement in holistic twig joins. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 784–791, 2005.

[11] P. B. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proceedings of the Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 281–291, 2001.

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

[13] M. Götz, C. Koch, and W. Martens. Efficient algorithms for the tree homeomorphism problem. In *DBPL*, pages 17–31, 2007.

[14] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. Van den Bussche. Database query processing using finite cursor machines. In *ICDT*, pages 284–298, 2007.

[15] M. Grohe, A. Hernich, and N. Schweikardt. Randomized computations on large data sets: Tight lower bounds. In *Proceedings of the 25th ACM Symposium on Principles of Database Systems (PODS)*, 2006. To appear.

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

[17] M. Grohe and N. Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *Proceedings of the 24th Symposium on Principles of Database Systems (PODS)*, pages 238–249, 2005.

[18] H. Jiang, H. Lu, and W. Wang. Efficient processing of XML twig queries with OR-predicates. In *Proceedings of the International Conference on Management of Data (SIGMOD)*, pages 59–70, 2004.

[19] H. Jiang, W. Wang, H. Lu, and J. Yu. Holistic twig joins on indexed XML documents. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, 2003.

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

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

[22] J. Lu, T. Chen, and T. W. Ling. Efficient processing of XML twig patterns with parent child edges: a look-ahead approach. In *Proceedings of the International Conference on Information and Knowledge Management (CIKM)*, pages 533–542, 2004.

[23] L. Qin, J. X. Yu, and B. Ding. TwigList : make twig pattern matching fast. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 850–862, 2007.

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

[25] T. Yu, T. W. Ling, and J. Lu. TwigStackList-: A holistic twig join algorithm for twig query with not-predicates on XML data. In *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, pages 249–263, 2006.

A Filtering lower bound

In Section 4.2, we proved Theorem 4.5 for the special case $Q = \texttt{//a/b}$. We now prove the more general result:

Theorem 4.5 (restated) *Let a, b be any two labels, and let Q be any basic twig query that contains the path segment //a/b. Furthermore, assume $a \neq b$ and that a, b do not appear elsewhere in Q. Then, for every algorithm for* FILTER_Q *and for every $d \geq 1$, there exists a document of depth at most $d - 1 + \text{depth}(Q)$, on which the algorithm uses at least $d - O(|Q| \log(|Q| \cdot d))$ bits of space.*

Proof. We first characterize the structure of Q: a schematic illustration of Q is presented in Figure 11. The *spinal path* of Q is the path from the root of Q (f_0) to b. Every T_i, for $i = 0, \ldots, k$ [or $i = a, b$], represents all the subtrees rooted at the children of f_i [or a, b]. T_i is essentially a forest, and may be empty. Note that the subtrees included in T_i can occur on either side of the spinal path.

Let $n = d$. We prove the theorem by showing an MDS reduction from the reverse-set-disjointness problem (RDISJ_n) to FILTER_Q. The MDS reduction is based on the following functions:

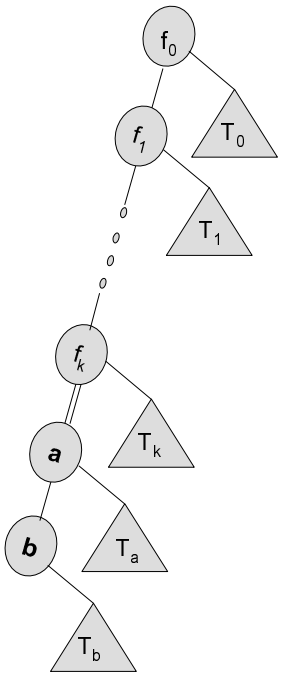

Figure 11: A schematic illustration of a basic twig query that contains a unique "//*a*/*b*".

- r^1_{in} and r^2_{in} construct the index streams T_a and T_b, respectively, of an XML document $D(x, y)$. The document structure, which is presented in Figure 12, is as follows. All edges denote parent-child relationships. $f_0, \ldots, f_k$ and $T_0, \ldots, T_k$ are exact copies of the corresponding elements in Q. $T_{a,i}$ and $T_{b,i}$ are an exact copy of T_a and T_b, respectively, for $i = 1, \ldots, n$. The only difference between documents of different (x, y) is the labels of the nodes $s_1, s_2, \ldots, s_n$ and $t_1, t_2, \ldots, t_n$. When $x_i = 1$, the corresponding node s_i is labeled 'a', and otherwise it is labeled c'. When $y_i = 1$, the corresponding node t_i is labeled 'b', and otherwise it is labeled 'd'. 'c' and 'd' are any labels that do not appear in Q.
- $r^1_c, \ldots, r^l_c$ construct the index streams of all the other labels in Q (except for T_a and T_b), of the document $D(x, y)$.
- $r_{out}(b) = b$ (note that the output of both RDISJ_n and FILTER_Q is one bit). Therefore $\text{DSS}(r_{out}) = 0$.

Claim A.1. $\text{DSS}(r^1_{in}) = \text{DSS}(r^2_{in}) \leq \log n$.

Proof. We describe algorithms A and B for r^1_{in} and r^2_{in}, resp. In order to output the next tuple in T_a [T_b], A [B] advances the stream x [y] to the next set bit. The tuple created is a simple function of the position of this bit. Specifically, if the position is i, then the index tuple of node s_i [t_i] is added to T_a [T_b].

It is easy to check that the index streams constructed are well-formed, i.e., sorted by the "Begin" attribute, and that they represent T_a and T_b of the document $D(x, y)$. The space needed for A [B] is $\log n$ bits for keeping the current position in x [y]. □

Claim A.2. $\forall i, 1 \leq i \leq l : \text{DSS}(r^i_c) = O(\log(|Q| \cdot n))$

Proof. Note that the document $D(x, y)$ has a fixed structure independent of x and y. In addition, the position of all the labels, except for a and b, is also fixed. It follows that the index stream of any label ($\neq a, b$) can be

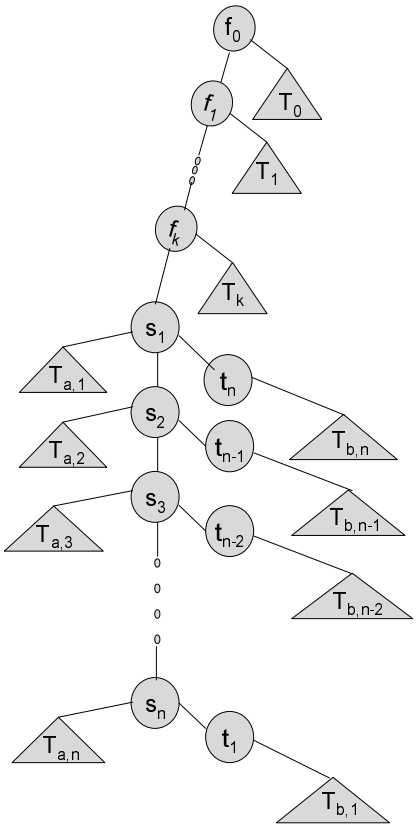

Figure 12: A schematic illustration of the document constructed by the reduction from RDISJ_n to FILTER_Q.

generated on-the-fly, based on the position of the required tuple. The space needed to maintain this "virtual" position while generating the index stream is $O(\log(|Q| \cdot n))$ bits. □

Lemma A.3. $\text{RDISJ}_n(x, y) = \text{FILTER}_Q(D(x, y))$.

Proof. First we assume that $\text{RDISJ}(x, y) = 1$. By definition, there exists some index $1 \leq i \leq n$, such that both x_i and y_i^R (i.e., y_{n+1-i}) are 1. This means that in $D(x, y)$ the label of s_i is 'a' and the label of t_{n+1-i} is 'b'. Since t_{n+1-i} is a child node of s_i, since s_i is a descendant of f_k, and since all other query nodes can be matched, the value of $\text{FILTER}_Q(D(x, y))$ is 1. The proof of the opposite direction is similar, and relies on the fact that a and b do not appear elsewhere in Q, and therefore can appear only in nodes s_* and t_*. □

Since $(r_{in}^1, r_{in}^2, r_c^1, \ldots, r_c^l)$ construct the index streams of $D(x, y)$, it follows that:

$$r_{out}(\text{FILTER}_Q(r_{in}^1(x), r_{in}^2(y), r_c^1(\epsilon), \ldots, r_c^l(\epsilon))) = \text{FILTER}_Q(D(x, y)) = \text{RDISJ}_n(x, y)$$

Therefore, by Corollary 4.4,

$$\begin{aligned}\text{MDSS}(\text{FILTER}_Q) &\geq \text{MDSS}(\text{RDISJ}_n) - \text{DSS}(r_{in}^1) - \text{DSS}(r_{in}^2) - \Sigma_{i=1}^{l}\text{DSS}(r_c^i) - \text{DSS}(r_{out}) \\ &\geq n - O(|Q|\log(|Q| \cdot n)) = d - O(|Q|\log(|Q| \cdot d))\end{aligned}$$

.

□