The Space Complexity of Processing XML Twig Queries Over Indexed Documents

Mirit Shalem #1, Ziv Bar-Yossef *2

#Department of Computer Science, Technion
Haifa, Israel
1mirit2s@cs.technion.ac.il

*Department of Electrical Engineering, Technion
Haifa, Israel
and Google Haifa Engineering Center
Haifa, Israel
2zivby@ee.technion.ac.il

***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.

I. Introduction

XQuery and XPath [1] 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* [2], 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., [2], [3], [4], [5], [6], [7], [8]). Much progress has been made in supporting wider fragments 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* [9], 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.

Table I 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

978-1-4244-1837-4/08/$25.00 © 2008 IEEE

TABLE I

SUMMARY OF OUR RESULTS

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
(/)*(//)?	$\tilde{O}(1)$	$\Omega(1)$	$\tilde{O}(out)$	$\Omega(out)$
(/)*(//)(//)*	$\tilde{O}(\min(d,out))$	$\Omega(\min(d,out))$	$\tilde{O}(out)$	$\Omega(out)$
(/|//)*(//)(/)(//|/)*	$\tilde{O}(d)$	$\Omega(d)$	$\tilde{O}(D)$	$\Omega(\max(d,out))$

D, d, and "out" denote the document size, its depth, and the output size, resp.

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]

[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.

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.

Due to lack of space, we feature in this extended abstract only an $\Omega(\text{out})$ lower bound for full-fledged evaluation[2] of tree queries and the new twig join algorithm. The full draft of this paper[3] discusses our other results: (1) the $\Omega(d)$ lower bound; (2) the $\Omega(\text{out})$ lower bound for pattern matching of tree queries (last column of the third table); (3) an $\Omega(\min(d, \text{out}))$ lower bound for pattern matching of certain path queries (second row of the third table); (4) extensions to our twig join algorithm and to the TwigStack algorithm [2] that are able to match the lower bounds we have in the pattern matching mode (first two rows of the third table). The $\tilde{O}(d)$ upper bound for path queries (in all evaluation modes) follows from the PathStack algorithm [2]. The $\tilde{O}(d)$ upper bound for filtering tree queries follows from the TurboXPath algorithm [10][4] (see also [8], [9]). 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.)

Our techniques. Space lower bounds for data stream algorithms are normally proved via communication complexity. It turns, however, that the standard communication complexity model 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 three problems: *delayed intersection*, *reverse set-disjointness* problem, and *tensor product* (the latter two are addressed in the full version). Our space lower bounds for twig query evaluation are obtained via reductions from these problems.

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",

[2]The $\Omega(\max(d, \text{out}))$ lower bound in the second table consists of two bounds: $\Omega(\text{out})$ and $\Omega(d)$. Here we present the proof of the $\Omega(\text{out})$ bound.

[3]Available from http://www.ee.technion.ac.il/people/zivby.

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

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.

II. Related Work

Starting with the seminal work of Bruno *et al.* [2] on holistic twig join algorithms, there have been many follow-up studies that presented improvements in the I/O and memory costs (e.g., [11], [5], [6], [4]) or extended the supported fragment of queries (e.g., [8], [7]). 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.* [12]. They state that any algorithm evaluating the query $//a[a$ and $a]$ requires super-constant memory.[5] 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.* [3] 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 [14], Segoufin [15], and Götz, Koch and Martens [16] studied the complexity of evaluating XPath queries over XML documents stored in main memory. Grohe, Koch, and Schweikardt [17] 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 [9], [18] 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.

The multiple-cursor multiple data stream model was analyzed in [19] and a lower bound for reverse set-disjointness was provided. Yet, the paper focuses on relational algebra queries and not on XPath.

III. 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.

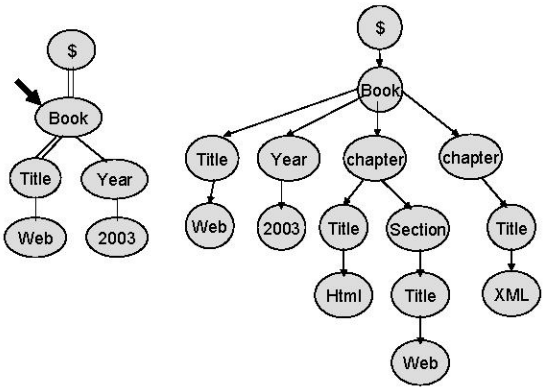

Fig. 1. Example XML document (right) and twig query (left) for the XPath query: $//Book[.//Title$="Web" and $Year$=2003].

XPath fragment. We focus on a fragment of XPath, which we call *basic twig queries*. Many existing algorithms focus on this type of queries [2], [6], [11]. 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[6]. 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.

An indexed XML document consists of a collection of index streams, one stream for every label that occurs in the

[5]While this statement is true, we suspect the proof included in [12] to be flawed, as it relies on a reduction to, and not from, evaluation of Select-Project-Join queries over continuous data streams [13].

[6]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.

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

IV. Ω(OUTPUTSIZE) LOWER BOUND

In this section we present the Ω(outputSize) space lower bound for full-fledged evaluation of tree queries that contain the (//)(/) pattern. 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. 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.

Theorem 4.1 (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.

The proof of the theorem appears in the full draft of the paper. 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 a reduction from the problem of *delayed intersection* in the *multiple data streams* 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.

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, 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 4.2: For any $n \geq 7$, the space complexity of DINT_n in the MDS model is at least $n - \log(n+1) - 4$.

The proof appears in Section V.

The reduction. We prove Theorem 4.1 for the case $Q =$ //z[b]/a by reducing DINT_n to FFE_Q. Let $n = S/2$. We prove that given an algorithm that solves FFE_Q on documents for which $|\text{FFE}_Q(D)|$ is at most S (i.e., $2n$) using k bits of space, we can design an algorithm that solves DINT_n with $k + O(\log n)$ bits of space. Theorem 4.1 would then immediately follow from Theorem 4.2.

In the reduction we use the input s, t, u, v to construct the index streams of an XML document $G(s,t,u,v)$ (see Figure 2). The document structure is the same for all inputs, 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'.

We now describe an algorithm A for DINT_n based on a given algorithm B for FFE_Q. Given the input vectors s, t, u, v, A simulates B on the document $G(s,t,u,v)$. To this end, A creates the two index streams T_a and T_b on the fly. The stream T_z is fixed and does not depend on the input. Whenever B needs to read a tuple from T_a (resp., T_b), A advances the cursor of $s \circ t$ (resp., $u \circ v$) until its next set bit, and "feeds" B with the corresponding tuple, which can be easily computed

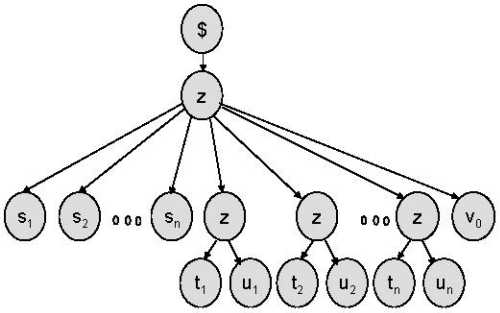

Fig. 2. The document $G(s,t,u,v)$.

based on the position of that bit. Whenever B outputs an a-element, i.e., an (s_i) or (t_i) element, A outputs "1" in the same position $(s \cap v^n)_i$ or $(t \cap u)_i$, respectively. It is easy to check that the index streams constructed are well-formed. Note also that the size of $\text{FFE}_Q(G(s,t,u,v))$ is at most $2n = S$, as required by B.

A uses only $k + O(\log n)$ bits: k bits for simulating B and $O(\log n)$ bits for keeping the positions of the two cursors. We next prove that this algorithm computes DINT_n correctly.

Proposition 4.3: 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. ■

V. 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 the lower bound for the delayed intersection problem (Theorem 4.2). We note that the same properties are used in proving the other space lower bounds included in the full draft of the paper. 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.

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. ■

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 $\text{SUCC}_{i,j}(x,y)$.

The following lemma shows that the packet sent by a player depends only on the prefix of the input seen by players along the communication path leading to this player (the proof, which is quite technical, appears at the full draft):

Lemma 5.6 (Suffix independence): Let $(x,y),(x',y') \in \text{PASS}(i,j)$. If $\text{PREF}_i(x) = \text{PREF}_i(x')$ and $\text{PREF}_j(y) = \text{PREF}_j(y')$, then $\text{PACKET}_{i,j}(x,y) = \text{PACKET}_{i,j}(x',y')$

Definition 5.7 (Prefix set): The *passing-prefix-set* of protocol P w.r.t. player (i,j), denoted $\text{PREF}(i,j)$, is the set: $\{(\text{PREF}_i(x), \text{PREF}_j(y)) \mid (x,y) \in \text{PASS}(i,j)\}$

The following is proved by induction, similarly to the proof of Lemma 5.6:

Proposition 5.8 (Rectangle property): $\forall(\alpha,\beta) \in \text{PREF}(i,j)$, and $\forall \gamma \in \{0,1\}^{m_1-i}$, $\delta \in \{0,1\}^{m_2-j}$, $(\alpha \circ \gamma, \beta \circ \delta) \in \text{PASS}(i,j)$.

The following bound on the size of the passing set derives from Proposition 5.8:

Corollary 5.9: $|\text{PASS}(i,j)| = |\text{PREF}(i,j)| \cdot 2^{(m_1-i)+(m_2-j)}$

The following is proved by induction, similarly to the proof of Lemma 5.6:

Proposition 5.10: 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.

Delayed intersection. We now use the TMC model to prove the space lower bound for delayed intersection in the MDS model.

Theorem 4.2 (restated) *For any $n \geq 7$, the space complexity of DINT_n in the MDS model is at least $n - \log(n+1) - 4$.*

Proof: We will prove that any 2-dimensional TMC protocol computing the delayed intersection problem has a max communication cost of at least $m = n - \log(n+1) - 4$ 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 $m < n - \log(n+1) - 4$. 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 Corollary 5.9, $|\text{PREF}(i,n-i)| = \frac{|\text{PASS}(i,n-i)|}{2^{2n+1}} \geq \frac{2^n}{n+1}$ (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. We exclude from $\text{PREF}(i,n-i)$ the four inputs, in which all bits are zero, except maybe for s_i and u_{n-i}. By the pigeonhole principle, one of these sets is of size at least $\frac{2^{n-3}}{n+1}$. Call this set A. Since the message sent by the congested player $(i,n-i)$ has at most m bits, where $m < n - \log(n+1) - 4$, and since this player has only two neighbors, the number of possible packets it can send is less than $\frac{2 \cdot 2^{n-4}}{n+1}$. There are $\frac{2^{n-3}}{n+1}$ different prefixes in A, therefore by the pigeonhole principle, there exist two pairs of prefixes $(s',u'),(s",u") \in A$ s.t. $\text{PACKET}_{i,n-i}(s',u') = \text{PACKET}_{i,n-i}(s",u")$, and $s'_i = s"_i$, and $u'_{n-i} = u"_{n-i}$.

Recall that we excluded zero prefixes, which means that the output depends on t and v, which have not been read yet. Therefore we know that no bit has been written to the output stream yet. We now define two different inputs for P, on one of which P must err. The above prefixes (s',u') and $(s",u")$ differ in at least one bit. First assume this is the k-th bit in s' and $s"$, where $k < i$.

Consider any strings $\alpha \in \{0,1\}^{n-i}$, $\beta \in \{0,1\}^i$, and $t \in \{0,1\}^n$. We next show that P must output the same answer on the two inputs: $(s' \circ \alpha, t, u' \circ \beta, 1)$ and $(s" \circ \alpha, t, u" \circ \beta, 1)$. Recall that $\text{PACKET}_{i,n-i}(s',u') = \text{PACKET}_{i,n-i}(s",u")$, and that the suffixes $s'_i \circ \alpha \circ t$ and $u'_{n-i} \circ \beta \circ 1$ are the same for the two executions. Therefore, by Proposition 5.10, P outputs the same value.

On the other hand, we now show that $\text{DINT}_n(s' \circ \alpha, t, u' \circ \beta, 1) \neq \text{DINT}_n(s" \circ \alpha, t, u" \circ \beta, 1)$. This would imply that P errs on at least one of the inputs. Recall that $\text{DINT}_n(s,t,u,v)$ is defined to be: $(s \cap v^n) \circ (t \cap u)$. Since in both inputs $v=1$, but they differ in the k-th bit in s, then their corresponding outputs also differ in the k-th bit in $(s \cap v^n)$.

The proof for the other case, where the two prefixes (s',u') and $(s",u")$ differ in the k-th bit in u' and $u"$, is very similar.

We choose a suffix for the two prefixes, such that the k-th bit in t is set. This way the value of the $(n+k)$-th bit in DINT_n of the two inputs is different, but the protocol outputs the same value. ■

VI. The Twig Join Algorithm

We now present our new constant space twig join algorithm for full-fledged evaluation of queries that do not contain the *(//)(/)* pattern. The main procedure of the algorithm, depicted in Figure 3, is Eval(Q, t, D), which gets as input a query 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 to which t is mapped by this match, and (ii) advances the t's cursor beyond e_t. 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* T_u corresponding to u is currently positioned[7], 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 relies on the special structure of Q and works by recursively searching for matches of the sub-queries rooted at the children of u. 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 T_u ($u = \text{parent}(v)$) is currently positioned, and returns true if and only if Q_v has a match in D_{e_u}, where e_v is a descendant of e_u whose relationship with e_u matches the axis of v. This procedure works by repeatedly advancing the cursor T_v, until finding the desired element e_v. If a match is found, the cursors of the corresponding nodes are advanced to positions that indicate the match.

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

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
```

Fig. 3. FFE algorithm (no *(//)(/)*)

The recursion depth of the algorithm equals the query depth. Each level requires space for storing $O(1)$ document elements. Therefore, the space complexity is $\tilde{O}(1)$. The running time of the algorithm is linear in the document size.

Example run. Consider the document and the query depicted in Figure 4. Suppose that a is the output node. Initially, the three cursors point to (a_1, b_1, c_1). The first iteration of Eval calls NextMatchUnderSelf($,$), which looks for an a node that has b and c descendants. The first a element to be checked is a_1. The call to NextMatchUnderSelf(a, a_1) advances the cursors T_b and T_c separately, until they 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 NextMatchUnderParent(b,a_2) and NextMatchUnderParent(c,a_2) return true, which means that NextMatchUnderParent(a,$) and NextMatchUnderSelf($, $) return true. Eval outputs a_2, advances T_a to a_3 and calls NextMatchUnderSelf again when the three cursors point to (a_3, b_3, c_1). Now NextMatchUnderSelf returns false, as there is no additional match, and Eval ends.

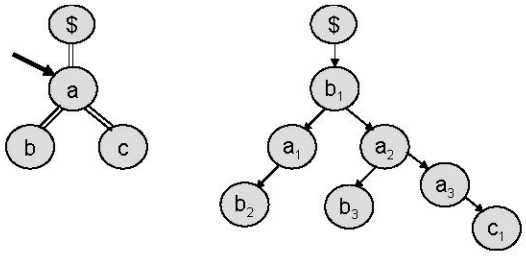

Fig. 4. An example XML document (right) and query (left). The query is *//a[.//b and .//c]*.

We now present a correctness analysis of the algorithm. We omitted several technical proofs, but the complete analysis appears in the full draft of this paper.

Cursor configurations. A notion that will play a crucial role in our analysis is *cursor configurations*. Let u be a query node. A *Q_u-cursor configuration* (or *Q_u-configuration*, in short) is a setting of the cursors $\{T_v\}_{v \in Q_u}$. 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, configurations encountered during an execution of an algorithm always dominate one another.

[7]As mentioned in Section III, 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 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.

Analysis of NextMatchUnderSelf. We start by analyzing the main subroutine, NextMatchUnderSelf. 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. In the following we prove that NextMatchUnderSelf is both *sound* (returns true, only if a match exists) and *complete* (if a match exists, returns true). We denote by Q and D any basic twig query and any document, respectively.

Lemma 6.1 (Soundness): If NextMatchUnderSelf(u, e_u) returns true, then its ending Q_u-configuration induces a match of Q_u in D_{e_u}.

Lemma 6.2 (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.

The proof of the soundness lemma is relatively easy and is done by induction on the height of u (see the full draft of the paper). We next prove the completeness lemma. We assume from now on that Q does not contain the *(//)(/)* pattern. The following propositions are a key to proving the lemma:

Proposition 6.3: Let ϕ be a match of Q in D. Then, for every child-axis node u, $\mathrm{depth}(\phi(u)) = \mathrm{depth}(u)$.

Proposition 6.4: If NextMatchUnderSelf(u, e_u) is called with a child-axis node u, then $\mathrm{depth}(e_u) = \mathrm{depth}(u)$.

The two proofs, which we omitted, are done by induction on $\mathrm{depth}(u)$.

Proof: [Proof of Lemma 6.2] We prove the lemma by induction on $k = \mathrm{height}(u)$. 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 $\mathrm{axis}(v)$.

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 6.5: 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 6.2. 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: [Proof of Claim 6.5] 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 $\mathrm{depth}(e''_a) < \mathrm{depth}(C'_v[a])$. Since C' induces a match, then according to Proposition 6.3, $\mathrm{depth}(C'_v[a]) = \mathrm{depth}(a)$, and thus $\mathrm{depth}(e''_a) < \mathrm{depth}(a)$. Therefore, based on Proposition 6.4, 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. ■

Analysis of Eval. In order to prove Eval is correct, we need to show it is *sound* (every element it outputs indeed matches t) and *complete* (every element that matches t node is output). We sketch below the proofs. The full details appear in the full draft of the paper.

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 6.1, 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.

Showing completeness is more intricate. Let $e_{t_1}, ..., e_{t_k}$ be the elements that match t, in document order. We prove that Eval outputs them in this order. To this end, 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. Due to the completeness property of NextMatchUnderSelf, we always move to the minimum match, and hence we are guaranteed not to miss a match of t with one of the e_{t_i}'s.

VII. 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 twig join algorithm that avoids this overhead, and achieves dramatic improvements in space for certain types of queries.

In the lower bound proofs, we make two assumptions about the model of query evaluation: (1) that every query node is associated with only a single cursor; and (2) that streams corresponding to labels that do not occur in the query are not accessed. In the full draft of this paper, we show how to eliminate the latter restriction when the label alphabet is sufficiently large. Overcoming the former restriction is more challenging and would require extension of our lower bound techniques or resorting to arguments similar to the ones used by Grohe *et al.* [19].

We focused on the dependence of the space complexity on parameters of the document, such as its depth and the output size. Query size may be an interesting factor to investigate in future work. Finally, empirical analysis of our algorithm could provide insights for its usefulness on real data.

Acknowledgments. This work was supported by the European Commission Marie Curie International Re-integration Grant.

References

[1] J. Clark and S. DeRose, "XML Path Language (XPath), Version 1.0," *W3C*, 1999, http://www.w3.org/TR/xpath.
[2] N. Bruno, N. Koudas, and D. Srivastava, "Holistic twig joins: optimal XML pattern matching," in *SIGMOD*, 2002, pp. 310–321.
[3] T. Chen, J. Lu, and T. W. Ling, "On boosting holism in XML twig pattern matching using structural indexing techniques," in *SIGMOD*, 2005, pp. 455–466.
[4] M. Fontoura, V. Josifovski, E. Shekita, and B. Yang, "Optimizing cursor movement in holistic twig joins," in *CIKM*, 2005, pp. 784–791.
[5] L. Qin, J. X. Yu, and B. Ding, "TwigList : make twig pattern matching fast," in *DASFAA*, 2007, pp. 850–862.
[6] J. Lu, T. Chen, and T. W. Ling, "Efficient processing of XML twig patterns with parent child edges: a look-ahead approach," in *CIKM*, 2004, pp. 533–542.
[7] T. Yu, T. W. Ling, and J. Lu, "TwigStackList-: A holistic twig join algorithm for twig query with not-predicates on XML data," in *DASFAA*, 2006, pp. 249–263.
[8] H. Jiang, H. Lu, and W. Wang, "Efficient processing of XML twig queries with OR-predicates," in *SIGMOD*, 2004, pp. 59–70.
[9] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, "On the memory requirements of XPath evaluation over XML streams." *J. Comput. Syst. Sci.*, vol. 73, no. 3, pp. 391–441, 2007.
[10] V. Josifovski, M. Fontoura, and A. Barta, "Querying XML streams," *The VLDB J.*, vol. 14, no. 2, pp. 197–210, 2005.
[11] H. Jiang, W. Wang, H. Lu, and J. Yu, "Holistic twig joins on indexed XML documents," in *VLDB*, 2003.
[12] B. Choi, M. Mahoui, and D. Wood, "On the optimality of holistic algorithms for twig queries," in *DEXA*, 2003, pp. 28–37.
[13] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom, "Characterizing memory requirements for queries over continuous data streams," *ACM Trans. Database Syst.*, vol. 29, no. 1, pp. 162–194, 2004.
[14] G. Gottlob, C. Koch, and R. Pichler, "The complex. of XPath query evaluation," in *PODS*, 2003, pp. 179–190.
[15] L. Segoufin, "Typing and querying XML documents: Some complex. bounds," in *PODS*, 2003, pp. 167–178.
[16] M. Götz, C. Koch, and W. Martens, "Efficient algorithms for the tree homeomorphism problem," in *DBPL*, 2007, pp. 17–31.
[17] M. Grohe, C. Koch, and N. Schweikardt, "Tight lower bounds for query processing on streaming and external memory data." in *ICALP*, 2005, pp. 1076–1088.
[18] Z. Bar-Yossef, M. Fontoura, and V. Josifovski, "Buffering in query evaluation over XML streams," in *PODS*, 2005, pp. 216–227.
[19] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. V. den Bussche, "Database query processing using finite cursor machines," in *ICDT*, 2007, pp. 284–298.