Tight Lower Bounds for Query Processing on Streaming and External Memory Data

Martin Grohe [a], Christoph Koch [b], and Nicole Schweikardt [a]

[a] *Institut für Informatik, Humboldt-Universität zu Berlin,*
Unter den Linden 6, D-10099 Berlin, Germany,
Email: {grohe/schweika}@informatik.hu-berlin.de

[b] *Database Group, Universität des Saarlandes,*
Postfach 15 11 50, D-66041 Saarbrücken, Germany,
Email: koch@cs.uni-sb.de

Abstract

It is generally assumed that databases have to reside in external, inexpensive storage because of their sheer size. Current technology for external storage systems presents us with a reality that performance-wise, a small number of sequential scans of the data is strictly preferable over random data accesses. Database technology — in particular query processing technology — has developed around a notion of memory hierarchies with layers of greatly varying sizes and access times. It seems that the current technologies scale up to their tasks and are very successful, but on closer investigation it may appear that our theoretical understanding of the problems involved — and of optimal algorithms for these problems — is not quite as developed.

Recently, data stream processing has become an object of study by the database management community, but from the viewpoint of database theory, this is really a special case of the query processing problem on data in external storage where we are limited to a single scan of the input data.

In the present paper we study a clean machine model for external memory and stream processing. We establish tight bounds for the data complexity of Core XPath evaluation and filtering. We show that the number of scans of the external data induces a strict hierarchy (as long as internal memory space is sufficiently small, e.g., polylogarithmic in the size of the input). We also show that neither joins nor sorting are feasible if the product of the number $r(n)$ of scans of the external memory and the size $s(n)$ of the internal memory buffers is sufficiently small, i.e., of size $o(n)$.

1 Introduction

It is generally assumed that databases have to reside in external, inexpensive storage because of their sheer size. Current technology for external storage systems (disks and tapes) presents us with a reality that a small number of *sequential scans* of the data is strictly preferable over random data accesses. Indeed, the combined latencies and access times of moving to a certain position in external storage are by orders of magnitude greater than actually reading a small amount of data once the read head has been placed on its starting position.

Database engines rely on main memory buffers for assuring acceptable performance. These are usually small compared to the size of the externally stored data. Database technology — in particular query processing technology — has developed around this notion of memory hierarchies with layers of greatly varying sizes and access times. There has been a wealth of research on query processing and optimization along these lines (cf. e.g. [37,16,44,28]). It seems that the current technologies scale up to current user expectations, but on closer investigation it may appear that our theoretical understanding of the problems involved — and of optimal algorithms for these problems — is not quite as developed.

Recently, data stream processing has become an object of study by the data management community (e.g. [17]) but from the viewpoint of database theory, this is, in fact, a special case of the query processing problem on data in external storage where we are limited to a single scan of the input data.

In summary, it appears that there are a variety of data management and query processing problems in which a comparably small but efficiently accessible main memory buffer is available and where accessing external data is costly and is best performed by sequential read/write scans. This calls for an appropriate formal model that captures the essence of external memory and stream processing. In this paper, we study such a model, which employs a Turing machine with one *external memory tape* (external tape for short) and an arbitrary number of *internal memory tapes* (internal tapes for short). The external tape initially holds the input; the internal tapes correspond to the main memory buffers of a database management system and are thus usually small compared to the input.

As computational resources for inputs of size n, we study the space $s(n)$ available on the internal tapes and the number $r(n)$ of scans of (or, random accesses to) the external tape, and we write $ST(r, s)$ to denote the class of all problems solvable by (r, s)-bounded Turing machines, i.e., Turing machines which comply to the resource bounds $r(n)$ and $s(n)$ on inputs of size n.

Formally, we model the number of scans, respectively the number of random accesses, by the number of reversals of the Turing machine's read/write head on the external tape. The number of reversals of the read/write head on the internal tapes remains unbounded. The reversals done by a read/write head are a clean and fundamental notion [9], but of course real external storage technology based on disks does not allow to reverse their direction of rotation. On the other hand, we can of course simulate k forward scans by $2k$ reversals in our machine model — and allowing for forward as well as backward scans makes our *lower* bound results even stronger.

As we allow the external tape to be both read and written to, this tape can be viewed, for example, as modeling a hard disk. By closely watching reversals of the external tape head, anything close to random I/O will result in a very considerable number of reversals, while a full sequential scan of the external data can be effected cheaply. We will obtain strong lower bounds in this paper that show that even if the external tape (whose size we do not put a bound on) may be written to and re-read, certain bounds cannot be improved upon. For our matching upper bounds, we will usually not write to the external tape. Whenever one of our results requires writing to the external tape, we will explicitly indicate this.

The model is similar in spirit to the frameworks used in [22,24], but differs from the previously considered *reversal complexity* framework [9]. Reversal complexity is based on Turing machines with a single read/write tape, where the overall number of reversals of the read/write head is the main computational resource. In our notion, only the number of reversals on the external tape is bounded, while reversals on the internal tapes are free; however, the space on the internal tapes is considered to be a limited resource. [1]

Apart from formalizing the $ST(r, s)$ model, we study its properties and locate a number of data management problems in the hierarchy of $ST(\cdot, \cdot)$ classes. Our technical contributions are as follows:

- We prove a reduction lemma (Lemma 4.1) which allows easy lower bound proofs for certain problems.
- We prove a hierarchy (Theorem 4.11), stating for each fixed number k that $k+1$ scans of the external memory tape are strictly more powerful than k

[1] The justification for this assumption is simply that accessing data on disks is currently about five to six orders of magnitude slower than accessing main memory. For that reason, processor cycles and main memory access times are often neglected when estimating query cost in relational query optimizers, where cost measures are often exclusively based on the amount of expected page I/O as well as disk latency and access times. Moreover, by taking buffer space rather than running time as a parameter, we obtain more robust complexity classes that rely less on details of the machine model (see also [43]).

scans of the external memory tape.

- We consider machines where the product of the number of scans of the external memory tape, $r(n)$, and internal memory tape size, $s(n)$, is of size $o(n)$, where n is the input size, and show that *joins* cannot be computed by (r, s)-bounded Turing machines (cf., Theorem 4.10). This also shows that for some *XQuery* queries, filtering is impossible for (r, s)-bounded machines with $r(T) \cdot s(T) \in o(n)$, where n is the size of the input XML document T.
- We show that the *sorting* problem cannot be solved by (r, s)-bounded Turing machines where $r(n) \cdot s(n) \in o(n)$ (cf., Theorem 4.7).
- We show (cf., Corollary 5.5) that for some *Core XPath* [14] queries, filtering is impossible for (r, s)-bounded machines with $r(T) \cdot s(T) \in o(d)$, where d denotes the *depth* of the input XML document T. This lower bound on *Core XPath* is *tight* in the following sense: there is an algorithm that solves the Core XPath filtering problem with a single scan of the external data (zero reversals) and $O(d)$ buffer space.

The primary technical machinery that we use for obtaining lower bounds is that of *communication complexity* (cf. [26]). Techniques from communication complexity have been used previously to study queries on streams [4,6,7,2,3,5,29,30,22]. The work reported on in [4] addresses the problem of determining whether a given relational query can be evaluated scalably on a data stream or not at all. In comparison, we ask for tight bounds on query evaluation problems, i.e. we give algorithms for query evaluation that are in a sense worst-case optimal. As we do, the authors of [6,7] study XPath evaluation; however, they focus on *instance data complexity* while we study worst-case bounds. Many of our results apply beyond stream processing in a narrow sense to a more general framework of queries on data in external storage. Also, our worst-case bounds apply for *any* evaluation algorithm possible, that is, our bounds are not in terms of complexity classes closed under reductions that allow for nonlinear expansions of the input (such as LOGSPACE) as is the case for the work on the complexity of XPath in [14,15,39].

Lower bound results for a machine model with *multiple* external memory tapes (or hard disks) are presented in [21,18]. In the present paper, we only consider a single external memory tape, and are consequently able to show (sometimes exponentially) stronger lower bounds.

The present paper is the full version of the conference contribution [19]. An informal overview of the methods used and results obtained in [19,21] can be found in [20].

2 Preliminaries

In this section we fix some basic notation concerning trees, streams, and query languages. We write $\mathbb{N}$ for the set of non-negative integers. If M is a set, then 2^M denotes the set of all subsets of M. Throughout this paper we make the following convention: Whenever the letters r and s denote functions from $\mathbb{N}$ to $\mathbb{N}$, then these functions are *monotone*, i.e., we have $r(x) \leqslant r(y)$ and $s(x) \leqslant s(y)$ for all $x, y \in \mathbb{N}$ with $x \leqslant y$.

2.1 Trees and Streams

We use standard notation for trees and streamed trees (i.e. *documents*). In particular, we write $\mathit{Doc}(T)$ to denote the XML document associated with an XML document tree T. An example is given in Figure 1.

Our precise notation concerning trees and streams is as follows:

Let τ be a finite set. We will use τ as a set of *tag names*. We associate with τ a finite alphabet Σ_τ as follows: For each symbol $a \in \tau$, the alphabet Σ_τ contains

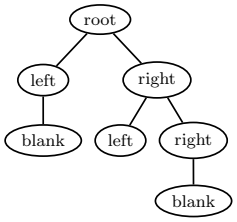
(i) a symbol $\langle a \rangle$ (corresponding to the opening tag labeled a), and
(ii) a symbol $\langle /a \rangle$ (corresponding to the closing tag labeled a).

Binary τ-trees are finite labeled ordered trees where each node has at most 2 children and is labeled with a symbol (i.e., tag name) in τ.

Unranked τ-trees are finite labeled ordered trees where each node may have an arbitrary number of children and is labeled with a symbol in τ. We use Trees_τ to denote the set of all unranked τ-trees. An unranked τ-tree T can be represented by a binary tree $\mathit{BinTree}(T)$ in a straightforward way by using the *first-child / next-sibling* notation (cf., e.g., the survey [33]).

The XML document $\mathit{Doc}(T)$ corresponding to an unranked τ-tree T can be viewed as a string over the alphabet Σ_τ, cf. Figure 1.

In particular, reading the string $\mathit{Doc}(T)$ from left to right corresponds to a *depth-first left-to-right traversal* of the tree T. For a set $\mathcal{T}$ of τ-trees we write $\mathit{Doc}(\mathcal{T})$ for the string language $\mathit{Doc}(\mathcal{T}) := \{ \mathit{Doc}(T) : T \in \mathcal{T} \} \subseteq \Sigma_\tau^*$. We use $\mathit{size}(T)$ to denote the number of nodes in T, and we use $\mathit{depth}(T)$ to denote the maximum number of edges on a path from the root to one of T's leaves.




```
<root>
  <left>
    <blank></blank>
  </left>
  <right>
    <left></left>
    <right>
      <blank></blank>
    </right>
  </right>
</root>
```

Fig. 1. A τ-tree T_1 and its XML document $Doc(T_1) \in \Sigma_\tau^*$ with tag names $\tau := \{\text{root}, \text{left}, \text{right}, \text{blank}\}$.

2.2 Query Languages

By $Eval(\cdot, \cdot)$ we denote the evaluation function that maps each tuple (Q, T), consisting of a query Q and a tree T to the corresponding query result. Let $\mathcal{Q}$ be a query language and let $\mathcal{T}_1 \subseteq Trees_\tau$ and $\mathcal{T}_2 \subseteq \mathcal{T}_1$. We say that $\mathcal{T}_2$ *can be filtered from* $\mathcal{T}_1$ *by a* $\mathcal{Q}$*-query* if, and only if, there is a query $Q \in \mathcal{Q}$ such that the following is true for all $T \in \mathcal{T}_1$: $T \in \mathcal{T}_2 \iff Eval(Q, T) \neq \emptyset$.

We assume that the reader is familiar with first-order logic (*FO*) and monadic second-order logic (*MSO*). To precisely fix a notion of *FO*- or *MSO*-definable queries over trees, one has to specify a way in which a tree $T \in Trees_\tau$ is represented by a logical structure. In the literature, several such representations have been considered (cf., e.g., [34,13,27]). With respect to *MSO*-definable queries it does not really matter which particular representation is chosen since they all lead to the same classes of *MSO*-definable queries. In the present paper we adopt the *first-child / next-sibling* representation used in a number of previous works (e.g. [33]), where a tree T is associated with a logical structure whose domain consists of the *nodes* of the tree, and which has two binary predicates *first-child* and *next-sibling* for connecting a node with its first child, respectively, with its next sibling, unary predicates *root, leaf, last-sibling* (with the obvious meanings), and a unary predicate $label_a$ for each tag name $a \in \tau$, for marking the nodes that are labeled with the symbol a.

An *FO*- or *MSO-sentence* (i.e., a formula without any free variable) specifies a Boolean query, whereas a formula with exactly one free first-order variable specifies a unary query, i.e., a query which selects a set of nodes from the underlying input tree.

It is well-known [10,41,42] that the MSO-definable Boolean queries on binary trees are exactly the (Boolean) queries that can be defined by finite (deterministic or nondeterministic) bottom-up tree automata. An analogous statement is true about MSO on unranked trees and unranked tree automata [8].

Theorem 4.10 in Section 4.4 gives a lower bound on the worst case complexity

of the language *XQuery*. As we prove a lower bound for *one* particular XQuery query, we do not give a formal definition of the language but refer to [45].

Apart from *FO*, *MSO*, and XQuery, we also consider a fragment of the XPath language, Core XPath. An example of a Core XPath query is

$$/\text{descendant::}*[\text{child::A and child::B}]/\text{child::}*,$$

which selects all children of descendants of the root node that (i.e., the descendants) have a child node labeled A and a child node labeled B. A complete formal definition of Core XPath can be found in [14]. Core XPath is a strict fragment of XPath, both syntactically and semantically. It is known that Core XPath is in LOGSPACE w.r.t. data complexity and P-complete w.r.t. combined complexity [15]. In [14], it is shown that Core XPath can be evaluated in time $O(|Q| \cdot |D|)$, where $|Q|$ is the size of the query and $|D|$ is the size of the XML data. Furthermore, every Core XPath query is equivalent to a unary MSO query on trees [13].

2.3 Communication complexity

To prove basic properties and lower bounds for our machine model, we use some notions and results from *communication complexity*, cf., e.g., [26].

Let A, B, C be sets and let $F : A \times B \to C$ be a function. In Yao's [46] basic model of communication, two players, Alice and Bob, jointly want to evaluate $F(x, y)$ for input values $x \in A$ and $y \in B$, where Alice only knows x and Bob only knows y. The two players can exchange messages according to some fixed protocol $\mathcal{P}$ that depends on F, but not on the particular input values x, y. The exchange of messages starts with Alice sending a message to Bob and ends as soon as one of the players has enough information on x and y to compute $F(x, y)$.

$\mathcal{P}$ is called a *k-round protocol*, for some $k \in \mathbb{N}$, if the exchange of messages consists, for each input $(x, y) \in A \times B$, of at most k rounds. The *cost* of $\mathcal{P}$ on input (x, y) is the number of bits communicated by $\mathcal{P}$ on input (x, y). The *cost of* $\mathcal{P}$ is the *maximal* cost of $\mathcal{P}$ over all inputs $(x, y) \in A \times B$. The *communication complexity of* F, *comm-compl*(F), is defined as the minimum cost of $\mathcal{P}$, over all protocols $\mathcal{P}$ that compute F. For $k \geqslant 1$, the *k-round communication complexity of* F, *comm-compl*$_k(F)$, is defined as the minimum cost of $\mathcal{P}$, over all k-round protocols $\mathcal{P}$ that compute F.

Many powerful tools are known for proving lower bounds on communication complexity, cf., e.g., [26]. In the present paper we will use the following basic lower bounds for the problem of deciding whether two sets are disjoint.

Definition 2.1 For $n \in \mathbb{N}$ let the function $Disj_n : 2^{\{1,..,n\}} \times 2^{\{1,..,n\}} \to \{0,1\}$ be given via

$$Disj_n(X,Y) := \begin{cases} 1\ , & \text{if } X \cap Y = \emptyset \\ 0\ , & \text{otherwise.} \end{cases}$$

For every $m \leqslant n$ we write $Disj_{n,m}$ to denote the restriction of $Disj_n$ to pairs of m-element subsets of $\{1,\ldots,n\}$.

Theorem 2.2 (cf., e.g., [26]) *For all $n \in \mathbb{N}$ and $m \leqslant n$, we have*

(a) $comm\text{-}compl(Disj_n) \geqslant n$.
(b) $comm\text{-}compl(Disj_{n,m}) = \Omega\Big(\log \binom{n}{m}\Big)$.
(c) $comm\text{-}compl(Disj_{m^2,m}) = \Omega(m \cdot \log m)$.

Proof: The proof of (a) is straightforward (cf., e.g., [26, Example 1.23]).
(b) is a result of Razborov [38] (see also [26, Example 2.12]).
(c) is an immediate consequence of (b), since $\log \binom{m^2}{m} \geqslant \log\Big(m^m\Big) = m \cdot \log m$. ■

3 Machine Model

We consider Turing machines with

(1) an input tape, which is a read/write tape and will henceforth be called "external memory tape" or "external tape", for short,
(2) an arbitrary number u of work tapes, which will henceforth be called "internal memory tapes" or "internal tapes", for short, and, if needed,
(3) an additional write-only output tape.

Let M be such a Turing machine and let ρ be a run of M. By $\mathrm{rev}(\rho)$ we denote the number of times the external memory tape's head changes its direction in the run ρ. For $i \in \{1,\ldots,u\}$ we let $\mathrm{space}(\rho, i)$ be the number of cells of internal memory tape i that are used by ρ.

3.1 The class $ST(r,s)$ for strings

Definition 3.1 ($\boldsymbol{ST}(r,s)$ for strings) Let $r : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$.

(a) A Turing machine M is (r, s)-*bounded*, if every run ρ of M on an input of length n satisfies the following conditions:
   (1) ρ is finite,
   (2) $1 + \mathrm{rev}(\rho) \leq r(n)$, and [2]
   (3) $\sum_{i=1}^{u} \mathrm{space}(\rho, i) \leq s(n)$, where u is the number of internal tapes of M.

(b) A string-language $L \subseteq \Sigma^*$ belongs to the class $ST(r, s)$ (resp., $NST(r, s)$), if there is a deterministic (respectively, nondeterministic) (r, s)-bounded Turing machine which accepts exactly those $w \in \Sigma^*$ that belong to L.

(c) A function $f : \Sigma^* \to \Sigma^*$ belongs to the class $ST(r, s)$, if there is a deterministic (r, s)-bounded Turing machine which produces, for each input string $w \in \Sigma^*$, the string $f(w)$ on its write-only output tape.

(d) We write $ST^{\vdash}(r, s)$ to denote the class of all problems in $ST(r, s)$ that can be solved by an (r, s)-bounded Turing machine *without ever writing* to the external memory tape.

For classes R and S of functions, we let

$$ST(R, S) := \bigcup_{r \in R, s \in S} ST(r, s).$$

If $k \in \mathbb{N}$ is a constant, then we write $ST(k, s)$ instead of $ST(r, s)$, where r is the function with $r(x) = k$ for all $x \in \mathbb{N}$. We freely combine these notations and use them for $ST^{\vdash}(\cdot, \cdot)$ and $NST(\cdot, \cdot)$ instead of $ST(\cdot, \cdot)$, too.

If we think of the external memory tape of an (r, s)-bounded Turing machine as representing the incoming stream, stored on a hard disk, then admitting the external memory tape's head to reverse its direction might not be very realistic. But as we mainly use our model to prove *lower* bounds, it does not do any harm either. We mainly see head reversals as a convenient way to simulate *random access*. Random access can be introduced explicitly into our model as follows: A *random access Turing machine* is a Turing machine M which has a special internal memory tape that is used as *random access address tape*, i.e., on which only binary strings can be written. Such a binary string is interpreted as a positive integer specifying an external memory address, that is, the position index number of a cell on the external tape (we think of the external tape cells being numbered by positive integers). The machine has a special state q_{ra}. If q_{ra} is entered, then in one step the external memory tape head is moved to the cell that is specified by the number on the random access address tape, and the content of the random access address tape is deleted.

[2] It is convenient for technical reasons to add 1 to the number $\mathrm{rev}(\rho)$ of changes of the head direction. As defined here, $r(n)$ bounds the number of sequential scans of the external memory tape rather than the number of changes of head directions.

Definition 3.2 Let $q, r, s : \mathbb{N} \to \mathbb{N}$. A random access Turing machine M is (q, r, s)-*bounded*, if it is (r, s)-bounded (in the sense of an ordinary Turing machine) and, in addition, every run ρ of M on an input of length n involves at most $q(n)$ random accesses.

Recall that the random access address tape is part of the internal memory of a random access Turing machine. Hence a (q, r, s)-bounded random access Turing machine only has random access to the first $2^{s(n)}$ memory cells of the external memory tape. Noting that a random access can be simulated with at most 2 changes of the direction of the external memory tape head, one immediately obtains:

Lemma 3.3 *Let $q, r, s : \mathbb{N} \to \mathbb{N}$. If a problem can be solved by a (q, r, s)-bounded random access Turing machine, then it can also be solved by an $(r + 2q, O(s))$-bounded Turing machine.*

In the subsequent parts of this paper, we will concentrate on ordinary Turing machines (without random access). Via Lemma 3.3, all results can be transferred from ordinary Turing machines to random access Turing machines.

3.2 The class ST(r, s) for trees

We make an analogous definition to $ST(r, s)$ for trees instead of strings:

Let τ be a set of tag names. Recall from Section 2 that Trees_τ denotes the set of all unranked τ-trees.

Definition 3.4 ($\boldsymbol{ST}(r, s)$ for trees)
Let $r : \mathit{Trees}_\tau \to \mathbb{N}$ and $s : \mathit{Trees}_\tau \to \mathbb{N}$.

(a) A Turing machine M is (r, s)-*bounded*, if every run ρ of M on an input string $\mathit{Doc}(T)$, for all $T \in \mathit{Trees}_\tau$ satisfies the following conditions:
   (1) ρ is finite,
   (2) $1 + \mathrm{rev}(\rho) \leq r(T)$,
   (3) $\sum_{i=1}^{u} \mathrm{space}(\rho, i) \leq s(T)$, where u is the number of internal tapes of M.

(b) A tree-language $\mathcal{T} \subseteq \mathit{Trees}_\tau$ belongs to the class $ST(r, s)$, if there is a deterministic (r, s)-bounded Turing machine M such that, for all $T \in \mathit{Trees}_\tau$, we have $T \in \mathcal{T}$ if, and only if, M accepts the string $\mathit{Doc}(T)$.

$\mathcal{T}$ belongs to the class $ST^{-}(r, s)$, if the Turing machine M does not ever write anything onto its external memory tape.

4 Lower bounds for the ST model

In this section, we prove our main lower bound results. In Subsection 4.1, we set up our general framework for deriving lower bounds for the ST model from communication complexity lower bounds. As an immediate consequence, in Subsection 4.2, we prove a tight lower bound for the *disjointness* problem. In the following two subsections, we apply this lower bound method to two problems of practical relevance, the *sorting* problem and the problem of computing *joins* in an XML context. We close this long section with a technical result, which establishes a strict hierarchy of complexity classes based on the number of head reversals.

4.1 A reduction lemma

The following lemma provides a convenient tool for showing that a problem L does not belong to $ST(r, s)$. The lemma's assumption can be viewed as a *reduction* from a communication problem to the problem L. The lemma's proof is based on the simple observation that during an (r, s)-bounded computation, only $O(r(n) \cdot s(n))$ bits can be communicated between the first and the second half of the external memory tape.

Lemma 4.1 *Let $\mathcal{N}$ be an infinite subset of $\mathbb{N}$ and let*

$$\Big(F_n : A_n \times B_n \to \{0,1\}\Big)_{n \in \mathcal{N}}$$

be a sequence of communication problems (defined on arbitrary finite sets A_n, B_n) for which a lower bound

$$\textit{comm-compl}(F_n) \;=\; \Omega(n)$$

holds.
Let Σ be an alphabet and let $\lambda : \mathbb{N} \to \mathbb{N}$ such that the following is true: For every $n \in \mathcal{N}$ there are functions $f_n : A_n \to \Sigma^*$ and $g_n : B_n \to \Sigma^*$ such that for all $X \in A_n$ and $Y \in B_n$ the string $f_n(X)g_n(Y)$ has length $\leqslant \lambda(n)$.

Then we have for all $r, s : \mathbb{N} \to \mathbb{N}$ with $r(\lambda(n)) \cdot s(\lambda(n)) \in o(n)$, that there is no (r, s)-bounded deterministic Turing machine which accepts a string of the

form $f_n(X)g_n(Y)$ *if, and only if,* $F_n(X,Y) = 1$.

Proof: For the sake of contradiction let us assume that there is an (r,s)-bounded Turing machine M which accepts a string of the form $f_n(X)g_n(Y)$ if, and only if, $F_n(X,Y) = 1$. Since M is (r,s)-bounded, on an input string of length N, M's internal memory tapes always have length $\leqslant s(N)$, and the external memory tape head can pass any particular external memory tape position p for at most $r(N)$ times.

Let $n \in \mathcal{N}$ and let X and Y be arbitrary elements from A_n and B_n, respectively. From the lemma's assumption we know that the string $f_n(X)g_n(Y)$ has length $N \leqslant \lambda(n)$ and that $f_n(X)g_n(Y) \in L$ if, and only if, $F_n(X,Y) = 1$.

In particular, any internal memory tape configuration during a run of M on $f_n(X)g_n(Y)$ can be represented by a bit-string of length $d \cdot s(\lambda(n))$, for a suitable constant d.

Let Q denote M's set of states. Using M, one obtains a communication protocol $\mathcal{P}_n$ that computes the function $F_n(\cdot,\cdot)$ as follows: Alice's input $X \in A_n$ is represented by the string $f_n(X)$, whereas Bob's input $Y \in B_n$ is represented by the string $g_n(Y)$. Let $p := |f_n(X)|$. Alice starts the protocol by starting the Turing machine M on input "$f_n(X)\cdots$" and letting it run until the first time M tries to access the external memory tape position $p{+}1$. Then she sends the current state and internal memory tape configuration of M to Bob. That is, she sends $\Big(\log|Q| + d \cdot s(\lambda(n))\Big)$ bits of information. Now, Bob has all the information needed to continue the execution of M on input "$\cdots g_n(Y)$" until the first time M tries to access the external memory tape position p. Then, Bob sends the current state and internal memory tape configuration of M to Alice. Alice and Bob continue in this manner until the Turing machine M stops, deciding whether or not $f_n(X)g_n(Y)$ belongs to L and hence providing one of the players with the desired information whether or not $F_n(X,Y) = 1$.

Since M passes the external memory tape position p for at most $r(\lambda(n))$ times, the above protocol $\mathcal{P}_n$ computes the function $F_n(\cdot,\cdot)$ by exchanging at most

$$r(\lambda(n)) \cdot \Big(\log|Q| + d \cdot s(\lambda(n))\Big)$$

bits of information. However, since $r(\lambda(n)) \cdot s(\lambda(n)) \in o(n)$, we can find, for every constant $c > 0$, an $n_0 \in \mathbb{N}$ such that, for every $n \geqslant n_0$,

$$r(\lambda(n)) \cdot \Big(\log|Q| + d \cdot s(\lambda(n))\Big) \quad < \quad c \cdot n.$$

Then, the above protocol $\mathcal{P}_n$ computes F_n with $o(n)$ bits of communication, contradicting the assumption that *comm-compl*$(F_n) = \Omega(n)$. ■

From the above reduction lemma and the communication bounds of Theorem 2.2 we immediately obtain:

Lemma 4.2 *Let Σ be an alphabet.*

(a) For every $n_0 \in \mathbb{N}$ let there be an $n \geqslant n_0$ and functions $f_n, g_n : 2^{\{1,..,n\}} \to \Sigma^*$ such that for all $X, Y \subseteq \{1,..,n\}$ the string $f_n(X)g_n(Y)$ has length $O(n)$.

Then, for all $r, s : \mathbb{N} \to \mathbb{N}$ with $r(n) \cdot s(n) \in o(n)$, there is no (r,s)-bounded deterministic Turing machine which accepts a string of the form $f_n(X)g_n(Y)$ if, and only if, $X \cap Y = \emptyset$.

(b) For every $m \in \mathbb{N}$ and $n := m \cdot \log m$ let there be functions f_n, g_n that map m-element subsets of $\{1,..,m^2\}$ to strings in Σ^*, such that for all m-element sets $X, Y \subseteq \{1,..,m^2\}$ the string $f_n(X)g_n(Y)$ has length $O(n) = O(m \cdot \log m)$.

Then, for all $r, s : \mathbb{N} \to \mathbb{N}$ with $r(n) \cdot s(n) \in o(n)$, there is no (r,s)-bounded deterministic Turing machine which accepts a string of the form $f_n(X)g_n(Y)$ if, and only if, $X \cap Y = \emptyset$.

Proof: *(a):* From Theorem 2.2(a) we know that $\mathit{comm\text{-}compl}(\mathit{Disj}_n) = \Omega(n)$. Therefore, Lemma 4.2(a) immediately follows from Lemma 4.1 for $F_n := \mathit{Disj}_n$ and $A_n := B_n := 2^{\{1,..,n\}}$.

(b): From Theorem 2.2(c) we know that

$$(*) \qquad \mathit{comm\text{-}compl}(\mathit{Disj}_{m^2,m}) \;=\; \Omega(m \cdot \log m).$$

We consider $\mathcal{N} := \{m \cdot \log m \;:\; m \in \mathbb{N}\}$ and let, for every $n = m \cdot \log m$,

$$A_n \;:=\; B_n \;:=\; \{Z \subseteq \{1,..,m^2\} \;:\; |Z| = m\}$$

and

$$F_n \;:=\; \mathit{Disj}_{m^2,m}.$$

From $(*)$ we know that $\mathit{comm\text{-}compl}(F_n) = \Omega(n)$. Now, Lemma 4.2(b) immediately follows from Lemma 4.1. ■

4.2 Disjointness

Every n-bit string $x = x_1 \cdots x_n \in \{0,1\}^n$ specifies a set

$$S(x) \;:=\; \{i : x_i = 1\} \;\subseteq\; \{1,..,n\}.$$

Let L_{Disj} consist of those strings $x\#y$ where x and y specify disjoint subsets of $\{1,\ldots,n\}$, for some $n \geqslant 1$. That is,

$$L_{Disj} := \Big\{ x\#y \ : \text{ exists } n \geqslant 1 \text{ with } x,y \in \{0,1\}^n \text{ and } S(x) \cap S(y) = \emptyset \Big\}.$$

From Lemma 4.2(a) one easily obtains:

Proposition 4.3 *Let $r : \mathbb{N} \to \mathbb{N}$ and $s : \mathbb{N} \to \mathbb{N}$.*
If $r(n) \cdot s(n) \in o(n)$, then $L_{Disj} \notin ST(r,s)$.

Proof: For every $n \in \mathbb{N}$ we choose functions $f_n, g_n : 2^{\{1,\ldots,n\}} \to \{0,1,\#\}^*$ as follows: For every $X \subseteq \{1,\ldots,n\}$ let $f_n(X) := x\#$ and $g_n(X) := x$, where $x = x_1 \cdots x_n \in \{0,1\}^n$ is the (unique) n-bit string with $S(x) = X$. Then, for all $n \in \mathbb{N}$ and all $X, Y \subseteq \{1,\ldots,n\}$ we have

$$f_n(X)g_n(Y) \in L_{Disj} \iff X \cap Y = \emptyset\,,$$

and $|f_n(X)g_n(Y)| = 2n+1$. Assuming that $r(n) \cdot s(n) \in o(n)$, we obtain from Lemma 4.2(a) that $L_{Disj} \notin ST(r,s)$. ■

Remark 4.4 *The bound given by Proposition 4.3 is tight, as it can be easily seen that $L_{Disj} \in ST(r,s)$ for all $r, s : \mathbb{N} \to \mathbb{N}$ with $r(n) \cdot s(n) \in \Omega(n)$. (If $s(n) \geqslant \log n$, then the Turing machine does not even need to write anything onto the external memory tape.)*

4.3 Sorting

In this subsection, we prove upper and lower bounds for the problem of sorting a given sequence of bitstrings. Formally, we identify sequences of (possibly empty) strings over the alphabet $\{0,1\}$ with strings over $\{0,1,\#\}$ and consider the function $F_{Sort} : \{0,1,\#\}^* \to \{0,1,\#\}^*$ defined by

$$F_{Sort}(x_1\# \ldots \#x_m) = x_{\pi(1)}\# \ldots \#x_{\pi(m)},$$

where $m \geqslant 0$, $x_1, \ldots, x_m \in \{0,1\}^*$, and π is a permutation of $\{1,\ldots,m\}$ such that $x_{\pi(1)} \leqslant \ldots \leqslant x_{\pi(m)}$ in the lexicographical order.

Observe first that F_{Sort} can trivially be solved by a $(1,n)$-bounded Turing machine which copies the entire input into its internal memory, sorts the input strings in internal memory, and then writes the sorted sequence onto its write-only output tape.

Concerning lower bounds let us first consider the decision problem associated with F_{Sort}, where the task is to decide whether the input sequence $x_1\# \cdots \# x_m$ is sorted in lexicographically ascending order: By applying a linear communication lower bound for the "(lexicographical) less-than"-predicate on bitsrings [46], it is easy to obtain a lower bound stating that even the restriction of this decision problem to inputs consisting of just two strings of length n cannot be solved by an (r, s)-bounded Turing machine with $r(n) \cdot s(n) \in o(n)$.

However, just deciding whether two long strings are sorted, may not be what we have in mind when we think about "the sorting problem". In the following, we show that with a little more effort we also obtain a tight linear lower bound for the problem of sorting many "short" strings. We always denote the size of the input string $x_1\# \ldots \# x_m$ by n, that is, we have $n = m - 1 + \sum_{i=1}^{m} |x_i|$.

For a function $\ell : \mathbb{N} \to \mathbb{N}$, we let F^{ℓ}_{Sort} be the restriction of F_{Sort} to input strings $x_1\# \ldots \# x_m$ with $|x_i| \leq \ell(n)$ for $1 \leq i \leq m$. Of course, any lower bound result for F^{ℓ}_{Sort} also holds for the general sorting problem F_{Sort}. We start with the following lower bound for F^{ℓ}_{Sort}.

Lemma 4.5 *Let $\ell, r, s : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq \ell(n) \geq 2 \log n$ and $r(n) \cdot s(n) \in o(n)$. Then F^{ℓ}_{Sort} cannot be computed by an (r, s)-bounded deterministic Turing machine.*

Proof: The proof is by a reduction from the disjointness problem $Disj_{m^2,m}$ to the sorting problem F^{ℓ}_{Sort}.

Let $\mathcal{N} = \{(k+1) \cdot 2^{k+2} - 1 \ : \ k \in \mathbb{N}\}$. For $n = (k+1) \cdot 2^{k+2} - 1 \in \mathcal{N}$ and $m = 2^k$, let A_n and B_n both be the set of all m-element subsets of $\{1, \ldots, m^2\}$ and $F_n = Disj_{m^2,m} : A_n \times B_n \to \{0, 1\}$. Then $comm\text{-}compl(F_n) = \Omega(m \cdot \log m) = \Omega(n)$ by Theorem 2.2(c).

For every $i \in \{1, \ldots, m^2\}$, let $b(i)$ be the binary representation of $i-1$ padded with zeroes to a string of length $2k$. We define functions $f_n : A_n \to \{0, 1, \#\}^*$ and $g_n : B_n \to \{0, 1, \#\}^*$ by

$$f_n(\{i_1, \ldots, i_m\}) = b(i_1)\, 0\, \# \ldots \# \, b(i_m)\, 0\# \ ,$$
$$g_n(\{i_1, \ldots, i_m\}) = b(i_1)\, 1\, \# \ldots \# \, b(i_m)\, 1 \ ,$$

where we assume that $i_1 < \ldots < i_m$. Then for all $X \in A_n$, $Y \in B_n$ the string $f_n(X)g_n(Y)$ has length $(2k + 1) \cdot 2m + 2m - 1 = (2k + 2) \cdot 2 \cdot 2^k - 1 = n$.

Hence by the Reduction Lemma 4.1, there is no $(r, 3s)$-bounded Turing machine that accepts a string $f_n(X)g_n(Y)$ if and only if the sets $X, Y \subseteq \{1, \ldots, m^2\}$ are disjoint (for all $k \in \mathbb{N}$ with $n = (k + 1) \cdot 2^{k+2} - 1$ and $m = 2^k$).

Now suppose for contradiction that F^{ℓ}_{Sort} can be computed by an (r, s)-bounded Turing machine S. We shall construct an $(r, 3s)$-bounded Turing machine T that on input $f_n(X)g_n(Y)$ decides whether the sets X, Y are disjoint. Let us write $f_n(X)g_n(Y)$ as $x_1\# \ldots \#x_m\#y_1\# \ldots \#y_m$ and observe that the length of the binary strings x_i, y_i is $(2k+1) \leq 2 \cdot \log n \leq \ell(n)$. Hence $f_n(X)g_n(Y)$ is an instance of F^{ℓ}_{Sort}.

The crucial observation is that the sets X and Y are disjoint if and only if there is no string z such that $z0, z1 \in \{x_1, \ldots, x_m, y_1, \ldots, y_m\}$. Furthermore, if there is such a string z, then $z0$ and $z1$ will be written on the output tape successively by the sorting machine.

Thus our machine T proceeds as follows: It simulates S; at any point it keeps the last two strings written by S to the output tape in internal memory. If during the course of the computation, it detects strings $z0$ and $z1$, it rejects (because then X and Y are not disjoint). Otherwise, it accepts after the simulation is completed.

Altogether, T is an $(r, 3s)$-bounded Turing machine which, on input $f_n(X)g_n(Y)$ decides whether the sets X and Y are disjoint. As argued above, however, such a machine cannot exist, and thus the proof of Lemma 4.5 is complete. ■

The next lemma provides an upper bound that matches the lower bound of Lemma 4.5.

Lemma 4.6 *Let $\ell, s : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geq \ell(n)$ and $s(n) \geq \log n$. Then F^{ℓ}_{Sort} can be computed by an $\big(n/s(n), O(s(n))\big)$-bounded deterministic Turing machine (that does not need to write anything onto its external memory tape).*

Proof: Suppose that the input is $x_1\# \ldots \#x_m$. Let n be the length of the input, $\ell := \ell(n)$, and $s := s(n)$. For pairwise distinct indices $i_1, \ldots, i_k \in \{1, \ldots, n\}$, we say that strings $x_{i_1}, \ldots, x_{i_k}$ *form an interval*, if $x_{i_1} \leq \ldots \leq x_{i_k}$ and for each $j \neq i_1, \ldots, i_k$ either $x_j \leq x_{i_1}$ or $x_j \geq x_{i_k}$. The *size* of the interval is $\sum_{j=1}^{k} |x_{i_j}|$.

We describe an $(n/s, O(s))$-bounded Turing machine that sorts the input. In each scan of the external memory tape, the machine reads an interval of size at least s into the internal memory, sorts it, and writes it to the output tape. This is done in such a way that the intervals of successive scans are successive.

To obey the memory restrictions, the scans are implemented as follows: After each scan, the machine stores the largest input string y written to the output tape so far and the number q of times y has been written to the output tape so far (remember that the input strings need not be distinct). Then in the

next scan, it successively reads new input strings into the internal memory, ignoring all input strings smaller than y and also the first q copies of y. As soon as the size of the strings in internal memory is at least $s+\ell$, the machine discards the currently largest string in internal memory whenever it finds a smaller one. Thus after a scan, there is an interval of size between s and $s+\ell$ in internal memory (except for the last scan, where the interval may be smaller). After the scan, the strings in internal memory are sorted and written to the output tape. A copy of the largest string y' and the number q' of times it has been written to the output tape so far is kept in the internal memory.

Clearly, n/s scans suffice. The internal memory size required is $O(s)$, because $s \geq \ell$ and $s \geq \log n$. The latter is needed for storing the multiplicities q. ■

In summary, Lemma 4.6 and Lemma 4.5 directly lead to the following theorem which, intuitively, states that sorting is possible if, and only if, the product of the number of head reversals and the internal memory size is at least as big as the input size.

Theorem 4.7 *Let $\ell, r, s : \mathbb{N} \to \mathbb{N}$ such that $s(n) \geqslant \ell(n) \geqslant 2 \log n$.*

(a) If $r(n) \cdot s(n) \in \Omega(n)$, then $F^{\ell}_{Sort} \in ST^{-}(r, s)$.
(b) If $r(n) \cdot s(n) \in o(n)$, then $F^{\ell}_{Sort} \notin ST(r, s)$.

It is straightforward to see that by using Merge-Sort, the sorting problem F^{ℓ}_{Sort} can be solved using $O(\log n)$ scans of external memory and internal memory of size $O(\ell(n))$, provided that *three* external memory tapes are available. In [21], this logarithmic bound is shown to be tight, for arbitrarily many external tapes. Note that Theorem 4.7 gives an exponentially stronger lower bound for the case of a *single* external memory tape.

4.4 Joins

Let τ be the set of tag names { rels, rel1, rel2, tuple, no1, no2, 0, 1 }. Recall from Section 2 that the alphabet Σ_τ consists of two letters, $\langle a \rangle$ and $\langle /a \rangle$, for each tag name $a \in \tau$. In the following, we will sometimes write $\langle a/ \rangle$ as abbreviation of the string $\langle a \rangle \langle /a \rangle$.

We represent a pair (A, B) of finite relations $A, B \subseteq \mathbb{N}^2$ as a τ-tree $T(A, B)$ whose associated XML document $Doc(T(A, B))$ is a Σ_τ-string of the following form: For each number $i \in \mathbb{N}$ let $Bin(i) = b^{(i)}_{\ell i} \cdots b^{(i)}_0$ be the binary representa-

tion of i. For each pair $(i,j) \in \{1,..,n\}^2$ let $Doc(i,j) :=$

$$\langle \text{tuple} \rangle \, \langle \text{no1} \rangle \, \langle b_{\ell_i}^{(i)} / \rangle \, \cdots \, \langle b_0^{(i)} / \rangle \, \langle / \text{no1} \rangle \, \langle \text{no2} \rangle \, \langle b_{\ell_j}^{(j)} / \rangle \, \cdots \, \langle b_0^{(j)} / \rangle \, \langle / \text{no2} \rangle \, \langle / \text{tuple} \rangle \, .$$

For each finite relation $A \subseteq \mathbb{N}^2$ let $t_1, .., t_{|A|}$ be the lexicographically ordered list of all tuples in A. We let $Doc(A) := Doc(t_1) \, \cdots \, Doc(t_{|A|})$. Finally, we let

$$Doc(T(A,B)) := \langle \text{rels} \rangle \; \langle \text{rel1} \rangle \; Doc(A) \; \langle / \text{rel1} \rangle \; \langle \text{rel2} \rangle \; Doc(B) \; \langle / \text{rel2} \rangle \; \langle / \text{rels} \rangle .$$

It is straightforward to see that the string $Doc(T(A,B))$ has length

$$O\Big((|A| + |B|) \cdot \log m\Big),$$

if $A, B \subseteq \{1, \ldots, m^2\}^2$.

We write $A \bowtie_1 B$ to denote the join of A and B on their first component, i.e., $A \bowtie_1 B := \{ \, (x,y) \; : \; \exists z \; A(z,x) \wedge B(z,y) \, \}$. We let

$$\begin{aligned} \mathcal{T}_{Rels} &:= \Big\{ T(A,B) \; : \; A, B \subseteq \mathbb{N}^2, \; A, B \text{ finite} \Big\} \\ \mathcal{T}_{EmptyJoin} &:= \Big\{ T(A,B) \in \mathcal{T}_{Rels} \; : \; A \bowtie_1 B = \emptyset \Big\} \\ \mathcal{T}_{NonEmptyJoin} &:= \Big\{ T(A,B) \in \mathcal{T}_{Rels} \; : \; A \bowtie_1 B \neq \emptyset \Big\} . \end{aligned}$$

Lemma 4.8 *$\mathcal{T}_{NonEmptyJoin}$ can be filtered from $\mathcal{T}_{Rels}$ by an XQuery query.*

Proof: We can choose the XQuery query $Q :=$

```
for $x in /rels/rel1/tuple/no1,
    $y in /rels/rel2/tuple/no1
where deep-equal($x,$y) return <tuple></tuple>
```

The first line of this query iteratively binds the variable `$x` to all subdocuments enclosed by `<rels><rel1><tuple><no1>` tags (and the corresponding closing tags), which encode the binary representation of the first component of the tuples in the first relation A. The second line iteratively binds the variable `$y` to all subdocuments enclosed by `<rels><rel2><tuple><no1>` tags, which encode the binary representation of the first component of the tuples in the second relation B. The third line returns `<tuple></tuple>`, if the two values are equal.

Hence the result of Q on the tree $T(A,B)$ consists of one "tuple"-node for each tuple in $A \bowtie_1 B$. In particular, $Eval(Q, T(A,B))$ is *empty* if, and only if, $A \bowtie_1 B = \emptyset$. ■

Lemma 4.9 *Let* $r, s : \mathit{Trees}_\tau \to \mathbb{N}$.
If $r(T) \cdot s(T) \in o\Big(\mathit{size}(T)\Big)$, *then* $\mathcal{T}_{\mathit{EmptyJoin}} \notin \mathit{ST}(r, s)$.

Proof: We use Lemma 4.2(b). For finite $X, Y \subseteq \mathbb{N}$ let

$$A_X \;:=\; \{\,(i,1) \,:\, i \in X\,\} \qquad \text{and} \qquad B_Y \;:=\; \{\,(i,2) \,:\, i \in Y\,\}\,.$$

Obviously, $A_X \bowtie_1 B_Y = \emptyset$ if, and only if, $X \cap Y = \emptyset$.

For every $m \in \mathbb{N}$ and $n := m \cdot \log m$ we choose functions $f_n, g_n : 2^{\{1,..,m^2\}} \to \Sigma_\tau^*$ via

$$\begin{aligned} f_n(X) \;&:=\; \langle\text{rels}\rangle\, \langle\text{rel1}\rangle\, \mathit{Doc}(A_X)\, \langle/\text{rel1}\rangle \\ g_n(Y) \;&:=\; \langle\text{rel2}\rangle\, \mathit{Doc}(B_Y)\, \langle/\text{rel2}\rangle\, \langle/\text{rels}\rangle\,. \end{aligned}$$

Then, for all m-element sets $X, Y \subseteq \{1, \ldots, m^2\}$, the string $f_n(X)g_n(Y) = \mathit{Doc}(T(A_X, B_Y))$ has length $O(m \cdot \log m) = O(n)$, and

$$f_n(X)g_n(Y) \in \mathit{Doc}(\mathcal{T}_{\mathit{EmptyJoin}}) \iff X \cap Y = \emptyset.$$

From Lemma 4.2(b) we obtain for arbitrary $r', s' : \mathbb{N} \to \mathbb{N}$ with $r'(n) \cdot s'(n) \in o(n)$ that there is no (r', s')-bounded Turing machine which accepts exactly those strings of the form $f_n(X)g_n(Y)$ where $X \cap Y = \emptyset$. Noting that

$$\mathit{size}(T(A_X, B_Y)) \;=\; O\Big(|\mathit{Doc}(T(A_X, B_Y))|\Big) \;=\; O(|f_n(X)g_n(Y)|) \;=\; O(n)\,,$$

one then obtains for arbitrary $r, s : \mathit{Trees}_\tau \to \mathbb{N}$ with $r(T) \cdot s(T) \in o\Big(\mathit{size}(T)\Big)$ that $\mathcal{T}_{\mathit{EmptyJoin}} \notin \mathit{ST}(r, s)$. ■

From Lemma 4.8 and Lemma 4.9 we immediately obtain a lower bound on the worst-case data complexity for filtering relative to an XQuery query:

Theorem 4.10 *The tree-language* $\mathcal{T}_{\mathit{NonEmptyJoin}}$

(a) can be filtered from $\mathcal{T}_{\mathit{Rels}}$ *by an XQuery query,*
(b) does not belong to the class $\mathit{ST}(r, s)$, *whenever* $r, s : \mathit{Trees}_\tau \to \mathbb{N}$ *with*

$$r(T) \cdot s(T) \;\in\; o\Big(\mathit{size}(T)\Big).$$

Let us note that the above bound is "almost tight" in the following sense: The problem of deciding whether $A \bowtie_1 B = \emptyset$ and, in general, every *FO*-definable problem belongs to $\mathit{ST}(1, n)$ — in its single scan of the external memory tape, the Turing machine simply copies the entire input on one of its internal

memory tapes and then evaluates the *FO*-sentence by the straightforward algorithm for *FO*-model-checking, which works with LOGSPACE w.r.t. data complexity (cf. e.g. [1]).

4.5 A hierarchy based on the number of scans

This subsection's main result is

Theorem 4.11 *For every fixed $k \in \mathbb{N}$ and all classes S of functions from $\mathbb{N}$ to $\mathbb{N}$ such that $O(\log n) \subseteq S \subseteq o\Big(\frac{\sqrt{n}}{(\lg n)^3}\Big)$ we have*

$$ST(k,S) \subsetneqq ST(k{+}1,S) \quad \textit{and} \quad ST^{-}(k,S) \subsetneqq ST^{-}(k{+}1,S).$$

The proof of this theorem is based on the following result due to Duris, Galil and Schnitger [11], who prove an exponential gap between k- and $k{+}1$-round communication complexity. They consider functions

$$f : \{0,..,2^m{-}1\} \to \{0,..,2^m{-}1\},$$

encoded as list of binary representations of the values $f(0), f(1), \ldots, f(2^m{-}1)$, and prove a lower bound on the k-round communication complexity of the language L_{k+1}, consisting of the encodings of functions f where

$$\underbrace{f(f(\cdots f(f}_{k+2}(0))\cdots)) \;=\; 2^m{-}1\,.$$

The precise definition of L_{k+1} is as follows:

Definition 4.12 For every $k \in \mathbb{N}$, let

$$\begin{aligned} L_{k+1} \;:=\; \{\, w_0 w_1 \cdots w_{2^m-1} \;:\; & m \in \mathbb{N},\ w_i \in \{0,1\}^m, \\ & \text{and there exist } j_1,..,j_{k+1} \text{ such that} \\ & w_0 = j_1, w_{j_i} = j_{i+1},\ w_{j_{k+1}} = 2^m{-}1 \ \}\,. \end{aligned}$$

Theorem 4.13 (Duris, Galil, Schnitger [11]) *For every $k \geqslant 1$, the following is true for all sufficiently large $n \in \mathbb{N}$:*

$$\begin{aligned} \textit{comm-compl}_{k+1}(F_{k+1,n}) &\leqslant (k{+}1)\cdot\log n\,, \quad \textit{but} \\ \textit{comm-compl}_{k}(F_{k+1,n}) &\geqslant \tfrac{\sqrt{n}}{36\cdot k^4\cdot(\log n)^3}\,, \end{aligned}$$

where the function $F_{k+1,n} : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$ *is given via*

$$F_{k+1,n}(x,y) := \begin{cases} 1\ , & \textit{if}\ \ xy \in L_{k+1} \\ 0\ , & \textit{otherwise.} \end{cases}$$

In fact, Duris et al. [11] prove an even stronger result, namely that their lower bound applies for all k-round protocols, even if communication complexity is measured as the minimum complexity over all arbitrary partitions of the input bits into two parts of equal size.

By using Theorem 4.13, we can show the following stronger variant of Theorem 4.11:

Theorem 4.14 *For every fixed* $k \geqslant 1$*,*

$$ST^{-}(k{+}1,\, O(\log n)) \ \cap\ NST(1,\, O(\log n)) \quad \not\subseteq \quad ST\Big(k, o\Big(\tfrac{\sqrt{n}}{(\log n)^3}\Big)\Big).$$

Proof: We use Theorem 4.13 and let

$$L'_{k+1} \ := \ \{\ 1^m \# w_0 \cdots w_{2^m-1} \ \ :\ \ m \in \mathbb{N},\ w_i \in \{0,1\}^m,\ w_0 \cdots w_{2^m-1} \in L_{k+1}\ \},$$

where L_{k+1} is the language fixed in Definition 4.12.

From the definition of L_{k+1} it is straightforward to see that L'_{k+1} belongs to $ST^{-}(k{+}1, O(\log n))$ — the Turing machine just has to store the current index $i \in \{0,..,k{+}1\}$ and the corresponding string w_{j_i} on its internal tapes and move the external tape head to the block of index $j_{i+1} := w_{j_i}$. To recognize L'_{k+1}, this requires at most k changes of the direction of the external tape head, internal space $O(\log k + \log n) = O(\log n)$ (since k is a constant), and no writing on the external memory tape.

A *nondeterministic* Turing machine with internal space $\Omega(\log n)$ does not even need a single reversal of the external tape head — it can simply guess the strings $w_{j_1}, .., w_{j_{k+1}}$ on one of its internal tapes and verify their "correctness" while scanning the external tape from left to right. This is possible in space $O(\log n)$, because k is constant and the length of each of the strings w_j is m, which is logarithmic in the input length $n = \Omega(2^m)$. Therefore, $L'_{k+1} \in NST(1, O(\log n))$

Assume, for the sake of contradiction, that $L'_{k+1} \in ST\Big(k, o\Big(\tfrac{\sqrt{n}}{(\log n)^3}\Big)\Big)$ via a Turing machine M that is (k,s)-bounded, for some function $s : \mathbb{N} \to \mathbb{N}$ with $s(n) \in o\Big(\tfrac{\sqrt{n}}{(\log n)^3}\Big)$. Then, in the same way as in the proof of Lemma 4.2, M leads to a k-round protocol $\mathcal{P}_n$, for all $n \in \mathbb{N}$, that computes the function

$F_{k+1,n}$ from Theorem 4.13 and has cost at most $d \cdot k \cdot s(n)$, for a suitable constant d (depending on M, but not on k or n). Since $s(n) \in o\Big(\frac{\sqrt{n}}{(\log n)^3}\Big)$, we can find sufficiently large n such that $d \cdot s(n) < \frac{\sqrt{n}}{36k^5(\log n)^3}$. Consequently, for such n we have $\mathit{comm\text{-}compl}_k(F_{k+1,n}) \leqslant d \cdot k \cdot s(n) < \frac{\sqrt{n}}{36k^4(\log n)^3}$, contradicting Theorem 4.13. This completes the proof of Theorem 4.14. ■

Finally, note that Theorem 4.11 is an immediate consequence of Theorem 4.14. Let us mention that a generalization of Theorem 4.11 from constants k to functions $r : \mathbb{N} \to \mathbb{N}$ can be found in [23].

Remark 4.15 On the other hand, of course, the hierarchy for the $ST^{\neg}$ classes collapses, if internal memory space is at least linear in the size of the input: For every $r : \mathbb{N} \to \mathbb{N}$ and for every $s : \mathbb{N} \to \mathbb{N}$ with $s(n) \in \Omega(n)$, we have

$$ST^{\neg}(r, s) \subseteq ST^{\neg}(1, n + s(n)) \quad \text{and} \quad ST^{\neg}(r, O(s(n))) = \mathrm{DSPACE}(O(s(n))).$$

Note, however, that the same statement for ST instead of $ST^{\neg}$ does not seem to be true, because during an (r, s)-bounded computation in which *writing* to the external memory tape is allowed, the external memory tape could get as long as $n + r(n) \cdot 2^{O(s(n))}$, which is too large for $\mathrm{DSPACE}(O(s(n)))$.

5 Tight bounds for filtering and query evaluation on trees

This section establishes tight bounds for the worst case data complexity of Core XPath evaluation and filtering.

5.1 Lower bound

We need the following notation: We fix a set τ of tag names via

$$\tau \;:=\; \Big\{ \text{root},\ \text{left},\ \text{right},\ \text{blank} \Big\}.$$

Let T_1 be the τ-tree from Figure 1. Note that T_1 has a unique leaf v_1 labeled with the tag name "left". For any arbitrary τ-tree T we let $T_1(T)$ be the τ-tree rooted at T_1's root and obtained by identifying node v_1 with the root of T and giving the label "left" to this node. Now, for every $n \geqslant 2$ let T_n be the τ-tree inductively defined via

$$T_n \;:=\; T_1(T_{n-1}).$$

It is straightforward to see that T_n has exactly $2n$ leaves labeled "blank". Let $x_1, .., x_n, y_n, .., y_1$ denote these leaves, listed in *document order* (i.e., in the order obtained by a pre-order depth-first left-to-right traversal of T_n). For an illustration see Figure 2.

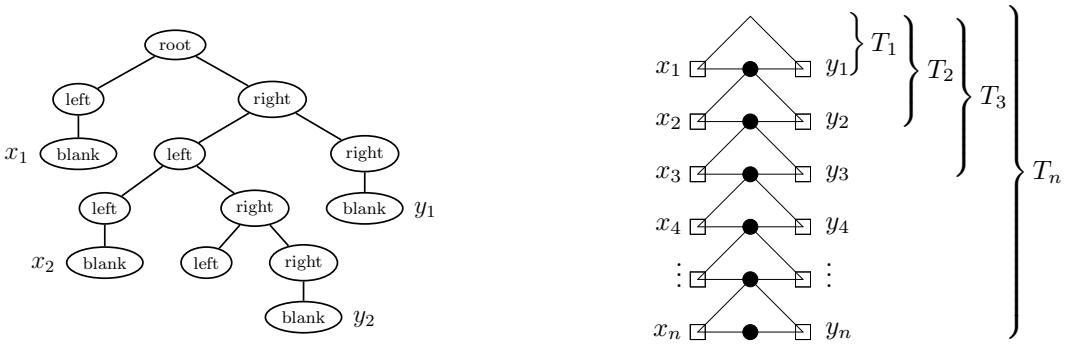


Fig. 2. Tree T_2 with nodes x_1, x_2, y_1, y_2 and tree T_n with nodes $x_1, .., x_n, y_1, .., y_n$.

We let $\tau_{01} := \tau \cup \{0, 1\}$. For all sets $X, Y \subseteq \{1, .., n\}$ let $T_n(X, Y)$ be the τ_{01}-tree obtained from T_n by replacing, for each $i \in \{1, .., n\}$,

- the label "blank" of leaf x_i by the label 1 if $i \in X$, and by the label 0 otherwise, and
- the label "blank" of leaf y_i by the label 1 if $i \in Y$, and by the label 0 otherwise.

We let

$$\mathcal{T}_{Sets} := \Big\{ T_n(X,Y) \ : \ n \geqslant 1,\ X, Y \subseteq \{1, .., n\} \Big\},$$
$$\mathcal{T}_{Disj} := \Big\{ T_n(X,Y) \in \mathcal{T}_{Sets} \ : \ X \cap Y = \emptyset \Big\},$$
$$\mathcal{T}_{NonDisj} := \Big\{ T_n(X,Y) \in \mathcal{T}_{Sets} \ : \ X \cap Y \neq \emptyset \Big\}.$$

Lemma 5.1 *(a) There is a Core XPath query Q such that the following is true for all τ-trees $T \in \mathcal{T}_{Sets}$: $Eval(Q, T) \neq \emptyset \iff T \in \mathcal{T}_{NonDisj}$.*
(b) There is a FO-sentence φ such that the following is true for all τ-trees T: $T \models \varphi \iff T \in \mathcal{T}_{NonDisj}$.

Proof: *(a):* We can choose $Q :=$

/descendant::∗[child::right/child::right/child::1]/child::left/child::1

which selects all nodes x that are labeled 1 and for which there exists a node z such that

(i) there exists a child z' of z which is labeled "left" such that x is a child of z',
(ii) there exists a child z'' of z which is labeled "right" and has a child z''' labeled "right" that has a child labeled 1.

It is straightforward to check that for all $T(X,Y) \in \mathcal{T}_{Sets}$ we have that $Q(T(X,Y))$ consists of exactly those nodes x_i for which both, x_i and y_i are labeled 1. I.e., $Q(T(X,Y)) = \{x_i : i \in X \cap Y\}$.

(b): It is known that all Core XPath queries are definable in *FO*, provided that trees are represented by structures where *descendant* and *following-sibling* (i.e., the transitive closures of *child* and of *next-sibling*) are available as binary relations (see Marx's paper [27] and the references therein). However, in Section 2 we chose to adopt the *first-child / next-sibling* representation (on which *FO* is strictly weaker than on the *descendant / following-sibling* representation), and therefore the proof of *(b)* requires a different (but easy) argument:

The desired *FO*-sentence φ is chosen as $\varphi := \chi \wedge \exists x\, \psi(x)$, where χ is a suitable *FO*-sentence expressing that the underlying tree T has the correct shape (among other things, χ stipulates that T is binary, i.e., each node of T has at most 2 children, that the first child of each node is labeled with one of the symbols *left*, 0, or 1, and the next sibling of each node is labeled with the symbol *right*). The *FO*-formula $\psi(x)$ is obtained as a straightforward formalization of the items (i) and (ii) in the proof of *(a)*. ■

Lemma 5.2 *Let $r, s : Trees_\tau \to \mathbb{N}$.*
If $r(T) \cdot s(T) \in o(depth(T))$, then $\mathcal{T}_{NonDisj} \notin ST(r,s)$.

Proof: We use Lemma 4.2(a). For every $n \in \mathbb{N}$, let p_n denote the position in the string $Doc(T_n)$ that carries the unique leaf of T_n carrying the label "left".

For every $n \in \mathbb{N}$ we choose functions $f_n, g_n : 2^{\{1,..,n\}} \to \Sigma_{\tau 01}^*$ as follows: For every $X \subseteq \{1,..,n\}$ let $f_n(X)$ be the prefix of $Doc(T_n(X,Y))$ up to position p_n, and let $g_n(Y)$ be the suffix of $Doc(T_n(X,Y))$ starting at position $p_n + 1$. Then, the string $f_n(X)g_n(Y) = Doc(T_n(X,Y))$ has length $10{\cdot}n + 1$, and

$$f_n(X)g_n(Y) \in Doc(\mathcal{T}_{Disj}) \iff X \cap Y = \emptyset.$$

From Lemma 4.2 we obtain for arbitrary $r', s' : \mathbb{N} \to \mathbb{N}$ with $r'(n) \cdot s'(n) \in o(n)$ that there is no (r', s')-bounded Turing machine which accepts exactly those strings of the form $f_n(X)g_n(Y)$ where $X \cap Y = \emptyset$.

Noting that

$$depth(T_n(X,Y)) \;=\; 2n+2 \;=\; O(|Doc(T_n(X,Y))|) \;=\; O(|f_n(X)g_n(Y)|)$$

one then obtains for arbitrary $r, s : Trees_\tau \to \mathbb{N}$ with $r(T) \cdot s(T) \in o(depth(T))$ that $\mathcal{T}_{NonDisj} \notin ST(r,s)$. ■

From Lemma 5.1 and Lemma 5.2 we directly obtain a lower bound on the worst-case data complexity of Core XPath filtering:

Theorem 5.3 *The tree-language $\mathcal{T}_{NonDisj}$*

(a) can be filtered from $\mathcal{T}_{Sets}$ by a Core XPath query,
(b) is definable by an FO-sentence (and therefore, also definable by a Boolean MSO query and recognizable by a tree automaton), and
(c) does not belong to the class $ST(r, s)$, whenever $r, s : Trees_\tau \to \mathbb{N}$ with $r(T) \cdot s(T) \in o(depth(T))$.

In the following subsection we match this lower bound with a corresponding upper bound.

5.2 Upper bounds

Recall from Section 2 and Definition 3.4 that, when provided as input for an (r, s)-bounded Turing machine, a tree $T \in Trees_\tau$ is represented by the XML document $Doc(T)$ (as indicated in Figures 1 and 3), and that a tree-language $\mathcal{T} \subseteq Trees_\tau$ belongs to the class $ST(r, s)$ if and only if there is an (r, s)-bounded Turing machine that accepts the XML documents $Doc(T)$ of exactly those τ-trees that belong to $\mathcal{T}$.

Further, recall that a tree-language $\mathcal{T} \subseteq Trees_\tau$ is definable by an *MSO*-sentence if, and only if, it is recognizable by an unranked tree automaton, respectively, if, and only if, the language $\{BinTree(T) : T \in \mathcal{T}\}$ of associated *binary* trees is recognizable by an ordinary (ranked) tree automaton (cf., e.g., [8,10,41,42]).

The following Theorem 5.4 shows that every tree-language $\mathcal{T}$ that is definable by an *MSO*-sentence, can be recognized by a Turing machine that performs a single (left-to-right) scan of its external memory tape and that requires internal memory of size linear in the depth of the (unranked) input tree. The Turing machine that we construct in the theorem's proof, in fact, corresponds to a pushdown automaton which simulates the run of a deterministic bottom-up tree automaton and uses its stack for keeping information about the path from the root to the currently visited node in the input tree. In the literature, similar kinds of pushdown automata have already been used for efficient query evaluation in various places, cf. e.g. [31,40].

Theorem 5.4 *Let $\mathcal{T} \subseteq Trees_\tau$ be a tree-language. If $\mathcal{T}$ is definable by an MSO-sentence (or, equivalently, recognizable by a ranked or an unranked finite tree automaton), then $\mathcal{T}$ belongs to $ST^-\Big(1, O(depth(\cdot))\Big)$.*

Proof: We proceed in two steps.
Step 1: We first show that $\mathcal{T}$ belongs to $ST^-\Big(2, O(depth(\cdot))\Big)$.

Let $\mathcal{B}$ be a deterministic bottom-up binary tree automaton (for an introduction see [42]) which accepts exactly the binary trees $\mathit{BinTree}(T)$ for $T \in \mathcal{T}$. In the following, we use the same notation as [25]; in particular, we assume that $\mathcal{B}$'s transition function has the form $\delta^{\mathcal{B}} : \Sigma \times (Q \cup \{\bot\}) \times (Q \cup \{\bot\}) \to Q$. Here, the special symbol $\bot$ is used as a "pseudo-state" for non-existent children.

In the following, we describe a Turing machine which on input $\mathit{Doc}(T)$ simulates the run of the automaton $\mathcal{B}$ on the binary tree $\mathit{BinTree}(T)$. Note that $\mathcal{B}$ starts at the leaves of $\mathit{BinTree}(T)$ and note that these leaves are conveniently accessible when reading the input XML document $\mathit{Doc}(T)$ from right to left (i.e., backwards).

We may assume that the input XML document $\mathit{Doc}(T)$ consists of a well-formed sequence of opening $\langle a \rangle$ and closing tags $\langle /a \rangle$, for tag symbols $a \in \tau$.[3] We evaluate $\mathcal{B}$ as follows, using a stack of states of the automaton $\mathcal{B}$. First we scan the input XML document $\mathit{Doc}(T)$ to the end (without doing anything). Then we reverse and scan it backwards. While scanning backwards, we do the following for each symbol $s \in \Sigma_\tau$ seen:

if s is a closing tag **then**
begin
  if the previously read symbol was a closing tag or
    there was no previous symbol (i.e., we are at the start of the backward scan) **then**
    push($\bot$);
end
else if s is an opening tag $\langle a \rangle$ **then**
begin
  if the previously read symbol was a closing tag **then**
    $q_1 := \bot$;
  else
    $q_1 :=$ pop();

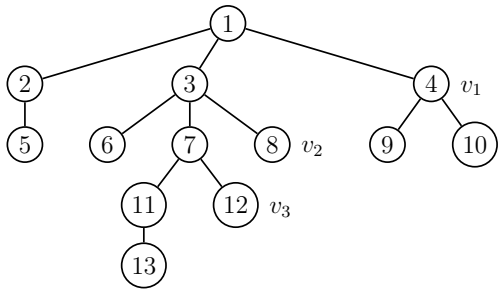
  $q_2 :=$ pop();
  $q := \delta^{\mathcal{B}}(a, q_1, q_2)$;
  push(q);
end

Consider the example run of Figure 3: Just after having processed the opening tag `<12>` of node v_3, the stack contains the symbols $\bot, \rho^{\mathcal{B}}(v_1), \rho^{\mathcal{B}}(v_2), \rho^{\mathcal{B}}(v_3)$ (where the final symbol is the top of the stack, and $\rho^{\mathcal{B}}(v)$ denotes the state assigned to node v by the run $\rho^{\mathcal{B}}$ of the tree automaton $\mathcal{B}$).

In general, it is easy to verify that whenever we are at a node v at depth d

[3] Non well-formed input can easily be detected by putting opening tags on the stack that we maintain in the algorithm below.

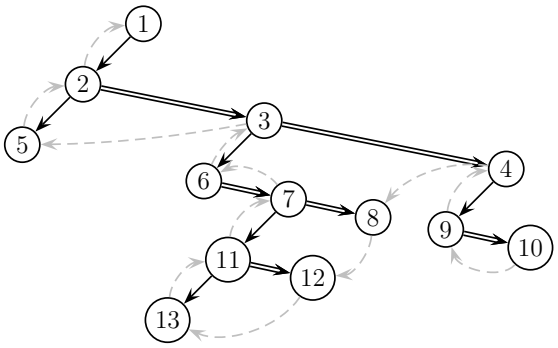
(a) An unranked tree T:



(b) The XML document $Doc(T)$:

```
<1><2><5></5></2><3><6></6><7><11><13></13></11><12></12></7><8></8></3><4><9></9><10></10></4></1>
```

(c) The binary first-child / next-sibling tree $BinTree(T)$:



Here, the *first-child* and the *next-sibling* relation are visualized by simple arcs $\longrightarrow$, respectively by double arcs $\Longrightarrow$. The dashed gray arcs indicate the traversal order for the evaluation of the bottom-up tree automaton $\mathcal{B}$ (starting at the node labeled 10)

Fig. 3. An unranked ordered tree T, its XML document $Doc(T)$, and the binary tree $BinTree(T)$.

in the unranked tree T (i.e., the Turing machine's external memory head is between the opening and the closing tag of v, and not between the opening and closing tag of a descendant of v), there are $d + 2$ items on the stack. Precisely, apart from the information on the tree automaton $\mathcal{B}$'s state at the child v' of v (in T) whose opening tag has just been read (in $Doc(T)$), the stack contains, for each node u on the path (in T) from the root to node v, information on $\mathcal{B}$'s state at the next sibling of u (respectively, the symbol $\perp$ if u does not have a next sibling).

Thus the depth of the stack never exceeds $depth(T)+2$. Since every stack entry consist of a single symbol, the space consumption of the internal memory tape is bounded by $depth(T) + 2$.

On termination of this $ST^{-}\Big(2, O\Big(depth(\cdot)\Big)\Big)$ algorithm, the stack will contain precisely one symbol, namely the tree automaton $\mathcal{B}$'s state at the root of $BinTree(T)$. If this state belongs to $\mathcal{B}$'s *accepting* states, then our Turing machine accepts; otherwise it rejects the input document $Doc(T)$.

Step 2: From $ST^{-}\Big(2, O(\mathit{depth}(\cdot))\Big)$ *to* $ST^{-}\Big(1, O(\mathit{depth}(\cdot))\Big)$.

Just as a binary bottom-up tree automaton on the *first-child / next-sibling* representation of (unranked) τ-trees can be computed, so can a binary tree automaton $\mathcal{B}$ be computed that works on a *last-child / previous-sibling* binary tree representation.

We can evaluate $\mathcal{B}$ in one single forward scan of the input by taking the algorithm of *Step 1*, and exchanging every occurrence of "opening tag" by "closing tag" and vice versa. Now we need only one forward scan to check whether $\mathcal{B}$ accepts.
Altogether, the proof of Theorem 5.4 is complete. ■

Recall that every Core XPath query is equivalent to a unary *MSO* query. Thus a Core XPath filter can be phrased as an MSO sentence on trees. From the Theorems 5.4 and 5.3 we therefore immediately obtain a tight bound for Core XPath filtering:

Corollary 5.5 *(a) Filtering from the set of unranked trees with respect to every fixed Core XPath query Q belongs to* $ST^{-}\Big(1, O(\mathit{depth}(\cdot))\Big)$.

(b) There is a Core XPath query Q such that, for all $r, s : \mathit{Trees}_\tau \to \mathbb{N}$ with $r(T) \cdot s(T) \in o\Big(\mathit{depth}(T)\Big)$, *filtering w.r.t. Q does not belong to $ST(r, s)$.*

Next, we provide an upper bound for the problem of computing the set $\mathit{Eval}(Q, T)$ of nodes in an input tree T matching a unary MSO (or Core XPath) query Q. We first need to clarify what this means, because writing the subtree of each matching node onto the output tape may require a very large amount of internal memory (or a large number of head reversals on the external memory tape), and this gives us no appropriate characterization of the difficulty of the problem. We study the problem of computing, for each node matched by Q, its index in the tree, in the order in which they appear in the document $\mathit{Doc}(T)$. We distinguish between the case where these indexes are to be written to the output tape in ascending order and the case where they are to be output in descending (i.e., reverse) order.

In [36] it is shown that *Boolean attribute grammars on ranked trees*, a formalism that captures the unary *MSO* queries, can be evaluated in two passes of the data. Independently, [25] describes a technique for evaluating unary MSO queries in two scans of the data. This technique is related to the nondeterministic version of query automata of [32,35] and the selecting tree automata of [12]. The first scan is a backward bottom-up tree automaton scan that writes the states computed for the nodes visited onto an external memory device, that the second scan, a forward scan during which a top-down deterministic tree automaton is evaluated, reads. In this sense, the following theorem

already was implicit in [36,25].

Theorem 5.6 *For every unary MSO or Core XPath query Q, the problem of computing, for input trees T, the nodes in $Eval(Q,T)$*

(a) in ascending order belongs to $ST\Big(3, O(depth(\cdot) + \log(size(\cdot)))\Big)$.
(b) in reverse order belongs to $ST\Big(2, O(depth(\cdot) + \log(size(\cdot)))\Big)$.

Proof: *(a):* Recall that every Core XPath query can be expressed as a unary *MSO* query. As explained above, every unary *MSO* query Q can be evaluated by a pair $\mathcal{A}$ and $\mathcal{B}$ of a bottom-up and a top-down deterministic tree automaton, where $\mathcal{A}$ runs on $BinTree(T)$ and $\mathcal{B}$ runs on the modified version of $BinTree(T)$ in which each node is labeled with $\mathcal{A}$'s state at this node. A node belongs to the result of the unary query Q, if $\mathcal{B}$'s state at this node belongs to a particular set of *selecting* states (for details see [25,12]).

Our Turing machine for evaluating Q on an input document $Doc(T)$ makes use of these automata $\mathcal{A}$ and $\mathcal{B}$ as follows: After scanning to the end of the input (without doing anything), it performs a backward scan during which it computes the run of $\mathcal{A}$, in the same way as described in the first part of the proof of Theorem 5.4. Now, however, on the external memory tape the corresponding states of $\mathcal{A}$ are stored by replacing the opening tag $\langle a \rangle$ of node v on the tape by a symbol $\langle a\ q = \rho^{\mathcal{A}}(v) \rangle$. (This is again a single tape symbol as both Σ and the state set of $\mathcal{A}$ are fixed.) At the end of this backward scan, the external memory tape contains, for each node v, the state $\rho^{\mathcal{A}}(v)$ computed by the run of $\mathcal{A}$ attached to it. Note that in the algorithm of the proof of Theorem 5.4, $\rho^{\mathcal{A}}(v)$ always gets available when the head on the external memory tape is on the position of the opening tag of node v, so we need no further buffer space besides the space occupied for the stack.

Then we perform a third scan, a forward scan during which we compute the run of $\mathcal{B}$. As $\mathcal{B}$ is a deterministic *top-down* tree automaton, the state of a node depends only on the state and the label of its parent. As $\mathcal{B}$ runs on the *first-child / next-sibling* representation of unranked trees, we have always $\rho^{\mathcal{B}}(v)$ available as soon as we have read the opening tag of node v. As described above, the state $\rho^{\mathcal{B}}(v)$ indicates whether v is in the query result. To be able to output the indexes of the selected nodes during the forward scan, we maintain a counter (initialized with 0) and during the scan, whenever we see an opening tag we increment it by one. Thus, whenever we decide that a node is part of the output, we write the current value of the counter — which is the index of the node in document order — to the output tape. This gives us the nodes matching the query in ascending order. To maintain this counter in internal memory, of course $\log(n)$ bits suffice, where n denotes the size of the input tree.

(b): Using the same ideas as in the proof of Theorem 5.4 (changing the automata from running on *first-child / next-sibling* to *last-child / previous-sibling* trees, we can compute the indexes of nodes matching a unary MSO query in *reverse order* (i.e., we output the node indexes while traversing the data backwards). ■

Note that this bound is tight: From Corollary 5.5(b) we know that, for some Core XPath query Q, not even *filtering* (i.e., checking whether $\mathit{Eval}(Q, T)$ is empty) is possible in $\mathit{ST}(r, s)$ if $r(T) \cdot s(T) \in o\Big(\mathit{depth}(T)\Big)$.

Remark 5.7 The proof of Theorem 5.6 requires (i) to scan the external memory tape both forward and backward, and (ii) to store states of the bottom-up automaton used in the proof construction of Theorem 5.4 on the external tape. If the query is considered fixed (data complexity), states are constant-size and can replace symbols of the input; but this means that we need to allocate space enough to store a state into each tape position of the input. The results of [25] only readily yield automata $\mathcal{A}$ whose state space is of size doubly exponential in the size of the given query (in the query language of the framework, monadic datalog). If we want to use this technique, we need Turing machines whose external tape alphabet is of size doubly exponential in the size of the given query.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of FOCS'04*, pages 540–549, 2004.

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

[4] A. Arasu, B. Babcock, T. Green, A. Gupta, and J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proceedings of PODS'02*, pages 221–232, 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of PODS'02*, pages 1–16, 2002.

[6] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. On the Memory Requirements of XPath Evaluation over XML Streams. In *Proceedings of PODS'04*, pages 177–188, 2004.

[7] Z. Bar-Yossef, M. Fontoura, and V. Josifovski. Buffering in Query Evaluation over XML Streams. In *Proceedings of PODS'05*, pages 216–227, 2005.

[8] A. Brüggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Non-ranked Alphabets: Version 1, April 3, 2001. Technical Report HKUST-TCSC-2001-05, Hong Kong Univ. of Science and Technology, 2001.

[9] J.-E. Chen and C.-K. Yap. Reversal Complexity. *SIAM Journal on Computing*, 20(4):622–638, 1991.

[10] J. Doner. Tree Acceptors and some of their Applications. *Journal of Computer and System Sciences*, 4:406–451, 1970.

[11] P. Duris, Z. Galil, and G. Schnitger. Lower bounds on communication complexity. *Information and Computation*, 73:1–22, 1987.

[12] M. Frick, M. Grohe, and C. Koch. Query evaluation on compressed trees. In *Proceedings of LICS'03*, pages 188–197, 2003.

[13] G. Gottlob and C. Koch. Monadic Datalog and the Expressive Power of Web Information Extraction Languages. *Journal of the ACM*, 51(1):74–113, 2004.

[14] G. Gottlob, C. Koch, and R. Pichler. Efficient Algorithms for Processing XPath Queries. In *Proceedings of VLDB'02*, pages 95–106, 2002.

[15] G. Gottlob, C. Koch, and R. Pichler. The Complexity of XPath Query Evaluation. In *Proceedings of PODS'03*, pages 179–190, 2003.

[16] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[17] T. J. Green, G. Miklau, M. Onizuka, and D. Suciu. Processing XML Streams with Deterministic Automata. In *Proceedings of ICDT'03*, Springer LNCS volume 2572, pages 173–189, 2003.

[18] M. Grohe, A. Hernich, and N. Schweikardt. Randomized Computations on Large Data Sets: Tight Lower Bounds. In *Proceedings of PODS'06*, pages 243–252, 2006.

[19] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *Proceedings of ICALP'05*, Springer LNCS volume 3580, pages 1076–1088, 2005.

[20] M. Grohe, C. Koch, and N. Schweikardt. The complexity of querying external memory and streaming data. In *Proceedings of FCT'05*, Springer LNCS volume 3623, pages 1–16, 2005.

[21] M. Grohe and N. Schweikardt. Lower bounds for sorting with few random accesses to external memory. In *Proceedings of PODS'05*, pages 238–249, 2005.

[22] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *External memory algorithms*, volume 50, pages 107–118. DIMACS Series In Discrete Mathematics And Theoretical Computer Science, 1999.

[23] A. Hernich and N. Schweikardt. Reversal Complexity Revisited. CoRR Report, arXiv:cs.CC/0608036, 7 Aug 2006. Available at `http://www.arxiv.org/abs/cs.CC/0608036`.

[24] J. E. Hopcroft and J. D. Ullman. Some results on tape-bounded Turing machines. *Journal of the ACM*, 16(1):168–177, 1969.

[25] C. Koch. Efficient Processing of Expressive Node-Selecting Queries on XML Data in Secondary Storage: A Tree Automata-based Approach. In *Proceedings of VLDB'03*, pages 249–260, 2003.

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

[27] M. Marx. First order paths in ordered trees. In *Proceedings of ICDT'05*, Springer LNCS volume 3363, pages 114–128, 2005.

[28] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, Springer LNCS volume 2625, 2003.

[29] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[30] S. Muthukrishnan. Data streams: algorithms and applications. In *Proceedings of SODA'03*, pages 413–413, 2003.

[31] A. Neumann and H. Seidl. Locating Matches of Tree Patterns in Forests. In *Proceedings of FSTTCS'98*, Springer LNCS volume 1530, pages 134–145, 1998.

[32] F. Neven. *Design and Analysis of Query Languages for Structured Documents – A Formal and Logical Approach*. PhD thesis, Limburgs Universitair Centrum, 1999.

[33] F. Neven. Automata Theory for XML Researchers. *SIGMOD Record*, 31(3):39–46, 2002.

[34] F. Neven and T. Schwentick. Expressive and Efficient Pattern Languages for Tree-Structured Data. In *Proceedings of PODS'00*, pages 145–156, 2000.

[35] F. Neven and T. Schwentick. Query automata over finite trees. *Theoretical Computer Science*, 275(1-2):633–674, 2002.

[36] F. Neven and J. van den Bussche. Expressiveness of Structured Document Query Languages Based on Attribute Grammars. *Journal of the ACM*, 49(1):56–100, 2002.

[37] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2002.

[38] A. A. Razborov. Applications of matrix methods to the theory of lower bounds in computational complexity. *Combinatorica*, 10:81–93, 1990.

[39] L. Segoufin. Typing and Querying XML Documents: Some Complexity Bounds. In *Proceedings of PODS'03*, pages 167–178, 2003.

[40] L. Segoufin and V. Vianu. Validating Streaming XML Documents. In *Proceedings of PODS'02*, pages 53–64, 2002.

[41] J. Thatcher and J. Wright. Generalized Finite Automata Theory with an Application to a Decision Problem of Second-order Logic. *Mathematical Systems Theory*, 2(1):57–81, 1968.

[42] W. Thomas. Languages, automata, and logic. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume III, pages 389–455. Springer, 1997.

[43] P. van Emde Boas. Machine Models and Simulations. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume 1, chapter 1, pages 1–66. Elsevier Science Publishers B.V., 1990.

[44] J. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[45] World Wide Web Consortium. XQuery 1.0 and XPath 2.0 Formal Semantics. W3C Working Draft (Aug. 16th 2002), 2002. `http://www.w3.org/XML/Query`.

[46] A. Yao. Some complexity questions related to distributive computing. In *Proceedings of STOC'79*, pages 209–213, 1979.