# Machine Models for Query Processing

Nicole Schweikardt

Institut für Informatik, Goethe-Universität Frankfurt am Main
schweika@informatik.uni-frankfurt.de

## 1 Introduction

The massive data sets that have to be processed in many application areas are often far too large to fit completely into a computer's internal memory. When evaluating queries on such large data sets, the resulting communication between fast internal memory and slower external memory turns out to be a major performance bottleneck.

Modern software and database technologies use clever heuristics to minimize the costs produced by external memory accesses. There has been a wealth of research on query processing and optimization along these lines (cf. e.g. [31, 16, 40, 27]), and it seems that the current technologies scale up to current user expectations.

Our theoretical understanding of the problems involved, however, is not quite as developed. Most results concerning the computational complexity of query languages are formulated in terms of classical complexity classes such as PTIME or PSPACE. Two examples of such results are that the combined complexity of evaluating relational algebra queries is PSPACE-complete [38, 37] and that the combined complexity of evaluating acyclic conjunctive queries belongs to PTIME [42] and is in fact LogCFL-complete [15]. The classes considered in computational complexity theory are usually based on Turing machines or random access machines, i.e., on machine models that do not take into account the existence of multiple storage media of varying sizes and access characteristics. Since the performance bottleneck for communicating between such storage media is completely ignored in classical complexity classes, "classical" results on the complexity of query evaluation only give a very rough measure of the complexity of query evaluation on a real computer.

In recent years, a number of machine models have been developed that take into account the existence of multiple storage media of varying sizes and access characteristics. These models are particularly useful for studying the complexity of query evaluation on massive data sets. The present paper gives an overview of such machine models. The two "extreme" models are the (very powerful) *external memory model*, presented in Section 2, and the (severely restricted) *mud model* (a model for massive, unordered, distributed computations), presented in Section 7. Further models considered in this survey are the *read/write streams* (a model for sequential external memory processing, Section 3), the *finite cursor machines* (a model for relational database query processing, Section 4), the *mpms-automata* (a model for processing indexed XML files, Section 5), and the *data stream model* (a model for processing data on-the-fly, Section 6).

As running examples throughout this article, we will take a closer look at the problem of *sorting* a given set of data items and at the problem of deciding whether two sets of data items are *disjoint*. Note that these two problems are of fundamental importance for query processing: Efficient query evaluation often relies on intermediate sorting steps; and already the easiest kind of *join* (i.e., the join of two *unary* relations) corresponds to computing the intersection of two sets.

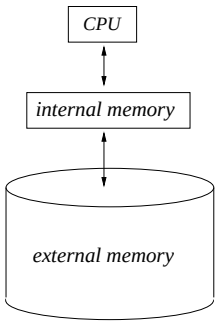
## 2 The External Memory Model

In the classical *random access machine* (RAM) model, the input consists of $N$ data items, each of which can be represented by a bitstring of length $O(\log N)$. An unbounded number of data items can be stored in memory. Access to any item in memory as well as arithmetics and bitwise operations on data items can be performed in constant time.

The basic *external memory model* (cf., e.g., [27]) can be viewed as a refinement of the RAM model where memory is divided into internal memory, capable of storing up to $M$ data items, and external memory of unbounded size. Data present in internal memory can be accessed quickly (as in the RAM model). Data present in external memory can only be accessed by an operation called *Input/Output communication* (I/O, for short) that moves a contiguous block of $B$ data items between internal and external memory. Initially, the $N$ input items are stored in external memory. One typically assumes that $M < N$ and $1 \leqslant B \leqslant M/2$. An illustration of the model is given in Figure 1.

The primary measure of performance in the external memory model is the number of I/O operations performed. Further relevant measures are (as in the RAM model) the total number of computation steps and the

**Database Principles Column.** Column editor: Leonid Libkin, School of Informatics, University of Edinburgh, Edinburgh, EH8 9AB, UK. E-mail: libkin@inf.ed.ac.uk.




$N =$ problem size (# data items)

$M =$ internal memory size (# data items)

$B =$ block transfer size (# data items)

$M < N, \quad 1 \leqslant B \leqslant M/2.$

Figure 1: The basic external memory model.

total amount of memory that is used.

This model is the most widely used model for designing and analyzing efficient external memory algorithms (cf., [27, 40]). On the one hand, it allows to design algorithms that explicitly control the communication between internal and external memory, so that algorithms that are "good" with respect to the model also perform well in practice. On the other hand, it is simple enough to allow for a high-level description of an algorithm and a rigorous mathematical analysis of its performance.

Obviously, *scanning*, i.e., reading the entire input in the order it is stored in external memory, takes

$$scan(N) \;=\; \Theta(N/B) \quad \text{I/O operations.}$$

A careful implementation of the *merge sort* algorithm shows that *sorting* can be performed using

$$sort(N) \;=\; \Theta\left(\tfrac{N}{B} \cdot \log_{M/B} \tfrac{N}{B}\right) \quad \text{I/O operations.}$$

Aggarwal and Vitter [2] proved a matching lower bound on the worst case complexity of sorting, provided that the machine model is restricted in such a way that the input data items are *indivisible* (i.e., they cannot be split up into their representations as bitstrings) and that at any point in time, the external memory consists of a permutation of the input items.

The *Parallel Disk Model*, introduced by Vitter and Shriver [41], is a generalization of the basic external memory model. In this model, the external memory is partitioned into $D$ independent disks, such that during each I/O operation, each of the $D$ disks can simultaneously transfer a block of $B$ contiguous data items into internal memory (one typically assumes that $1 \leqslant D{\cdot}B \leqslant M/2$). Furthermore, instead of a single CPU there are $P$ identical processors that work in parallel and that are connected by a network. Each of these processors has access to internal memory of size $M/P$. Furthermore, if $D \geqslant P$, then each processor owns $D/P$ of the $D$ disks, and if $D < P$, then each disk is shared by $P/D$ processors.

It is obvious that *scanning* can be done by performing $\Theta(N/DB)$ I/O operations. Concering *sorting*, an elaborate construction by Nodine and Vitter shows the following:

**Theorem 2.1 ([29])** *Sorting can be performed in the Parallel Disk Model using* $O\left(\frac{N}{DB} \cdot \log_{M/B} \frac{N}{B}\right)$ *I/O operations.*

The algorithm treats data items as indivisible units, and at any point in time during the execution of the algorithm, the content of external memory consists of a permuation of the input. For this restricted version of the Parallel Disk Model, a matching lower bound for sorting was obtained by Aggarwal and Vitter in [2].

**Suggestions for further reading:** An overview of external memory algorithms for various computation problems is given in [39]; more details can be found in the survey [40] and the books [1, 27].

**An important future task:** Concerning the external memory model and its extensions, a number of lower bound results are known, among them the lower bound for *sorting* mentioned above. A common feature of these lower bounds is that they rely on the assumption that the input data items are *indivisible* and that at any point in time the external memory consists, in some sense, of a permutation of the input items. It is a challenging future task to develop methods for proving lower bounds for the external memory model or the Parallel Disk Model without relying on such an indivisibility assumption.

# 3 Read/Write Streams

The external memory model considered in the previous section distinguishes between accesses to internal memory and accesses to external memory. Current technology for external storage systems (disks and tapes), however, presents us with a situation where *sequential scans* are strictly preferable over *random accesses* to external memory: A random access to a hard disk requires to move the read/write head to a certain position of the disk, and this is a comparably slow mechanical operation. During the time required for a single such *random* access, a considerable amount of data stored *in sequence*, starting at the current position of the disk's read/write head could have been processed. Modern software and database technologies use clever heuristics to minimize the number of accesses and to prefer streaming over random accesses to external memory.

The present section concentrates on a machine model for external memory processing which was introduced in [21, 22] and which has available

- *internal memory* that can be accessed very fast, but that is limited in size, and, additionally,
- *external memory* that can store huge amounts of data, which can easily be accessed (for reading as well as for writing) in a sequential way, but for which *random accesses* are expensive.

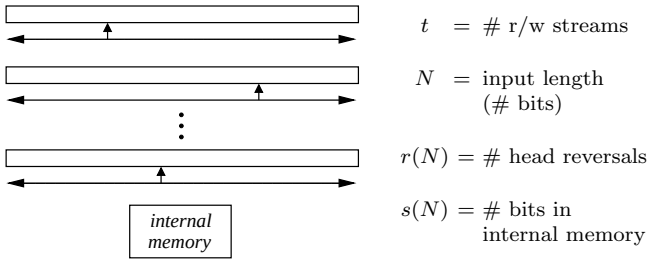



Figure 2: The model of read/write streams.

Formally, this model is based on a standard multi-tape Turing machine. Some of the tapes of the machine, among them the input tape, represent external memory (each of these tapes can be viewed as representing an external memory device such as, e.g., a hard disk). They are unrestricted in size, but access to these tapes is restricted by allowing only a certain number $r(N) - 1$ of reversals of the head directions (where $N$ denotes the size of the input, measured in terms of the number of bits necessary for representing the entire input). This may be seen as a way of (a) restricting the number of sequential scans and (b) restricting random accesses to these tapes (because each random access can be simulated by moving the head to the desired position on a tape, and this involves at most two head reversals). The remaining tapes of the Turing machine represent internal memory. Access to these internal memory tapes is unrestricted, but their size is bounded by a parameter $s(N)$. Such Turing machines are called $(r, s, t)$-*bounded*, if $t$ is the total number of external memory tapes of the machine.

It sometimes is convenient to adopt Beame, Jayram, and Rudra's [7] informal view of this as being a computation model where, in addition to some internal memory, a constant number of *read/write streams* (r/w, for short) are available; each such r/w stream corresponds to an external memory tape of the Turing machine. An illustration of the model is given in Figure 2. The resources of interest are

(1) the number $r(N)$ of scans of (or, head reversals on) the r/w streams (where $N$ denotes the number of bits necessary for representing the entire input),

(2) the size $s(N)$ of internal memory, and

(3) the total number $t$ of r/w streams that are available.

In the remainder of this article, we call this the *computational model of r/w streams*, and we sometimes informally say that there is an $(r, s, t)$-*bounded algorithm on r/w streams* for a specific problem, iff this problem can be solved by an $(r, s, t)$-bounded Turing machine.

When considering problems that produce an output other than just the answer "yes" or "no", we assume that there is an additional write-only output stream available.

**Remark 3.1** Note that this model allows *forward* scans as well as *backward* scans of the external memory tapes. Even more, it allows the r/w heads to reverse direction in the middle of a tape. While it is realistic to assume that in current "real-world" hard disks, sequential *forward* scans can be performed very efficiently, this is not the case for *backward* scans. Thus, when aiming at designing efficient algorithms for r/w streams, one should make sure that the algorithms essentially use only forward scans of the r/w streams. On the other hand, when aiming at *lower bounds*, i.e., showing that some problem can *not* be solved by any $(r, s, t)$-bounded algorithm on r/w streams, allowing for forward scans as well as backward scans makes lower bound results only stronger.

Having in mind the motivation that $(r, s, t)$-bounded algorithms on r/w streams serve as a computation model for external memory processing, we are mainly interested in cases where the size $s(N)$ of the internal memory is considerably smaller than the input size $N$, and the number $r(N)$ of sequential scans of (or, correspondingly, the number of random accesses to) external memory is, preferably, as small as possible.

Note that a priori there is no restriction on the running time or the size of the external memory of an $(r, s, t)$-bounded Turing machine. However, an easy induction on $r(N)$ shows that implicitly these two parameters are bounded in terms of the number of head reversals, the internal memory space, and the input size: Every run of an $(r, s, t)$-bounded Turing machine on an input of length $N$ consists of at most $N \cdot 2^{O(r(N)\cdot(t+s(N)))}$ computation steps (cf., [20]).

There are substantial differences between the computational power of the model where only *one* r/w stream is available and the model where $t \geqslant 2$ r/w streams are available. Both scenarios will be considered in the next two subsections.

## 3.1 One r/w stream

Let us first consider the problem *short-sorting*, which is the restriction of the sorting problem to inputs that consist of a list of an arbitrary number $m$ of bitstrings $x_1, \ldots, x_m$, each of which has length at most $2 \log m$. Note that the total length of the input is $N = \Theta(m \log m)$.

It can be easily seen that, for any function $s$ with $s(N) \geqslant 4 \log N$, the problem *short-sorting* can be solved by an $(r, s, 1)$-bounded algorithm on one r/w stream with $r(N) = O(N/s(N))$ sequential scans and internal memory of size $s(N)$. A matching lower bound was proved in [21], leading to the following result.

**Theorem 3.2 ([21])** *Let $r, s$ be functions such that $s(N) \geqslant 4 \log N$. The problem short-sorting can (respectively, cannot) be solved by an $(r, s, 1)$-bounded algorithm on one r/w stream if $r(N) \cdot s(N)$ is of size $\Omega(N)$ (respectively, of size $o(N)$).*

The basic idea for proving a lower bound on resources necessary for solving a problem in the computation model with one r/w stream is to divide the r/w stream into two parts. Obviously, during an $(r, s, 1)$-bounded computation on an input of size $N$, the border between the two parts of the r/w stream can be crossed at most $r(N)$ times. Each time we cross this border, we can only "transport" the amount of information that is currently stored in internal memory, and this consists of at most $O(s(N))$ bits. Consequently, during an entire $(r, s, 1)$-bounded computation, only $O(r(N) \cdot s(N))$ bits of information can be communicated between the two parts of the r/w stream. Therefore, lower bounds known from *communication complexity* (cf., [26]) almost immediately imply corresponding lower bounds for $(r, s, 1)$-bounded algorithms on one r/w stream.

Using this, one for example obtains that the *set disjointness* problem cannot be solved by an $(r, s, 1)$-bounded algorithm on one r/w stream if $r(N) \cdot s(N)$ is of size $o(N)$. Here, the *set disjointness* problem receives as input two lists of bitstrings $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$, and the task is to decide whether $\{x_1, \ldots, x_m\} \cap \{y_1, \ldots, y_m\} = \emptyset$.

By using this lower bound, one immediately obtains that evaluating *relational algebra queries* is hard for algorithms on one r/w stream: Note that if $A$ and $B$ are unary relations, then $A \bowtie B = A \cap B$. Therefore, already the basic task of checking whether the join of two relations $A$ and $B$ is empty cannot be performed if $r(N) \cdot s(N)$ is of size $o(N)$ (where $N$ denotes the number of bits used for storing the two input relations $A$ and $B$).

In a similar way one also obtains lower bounds for evaluating queries against XML data. For example, two unary relations $A$ and $B$ can easily be encoded by an XML document, such that the query which asks if the join of $A$ and $B$ is empty can be formalized in the language *XQuery*. This immediately leads to the result stating that there exists an *XQuery* query $Q$ such that, for all functions $r, s$ with $r(N) \cdot s(N) = o(N)$, there is no $(r, s, 1)$-bounded algorithm on one r/w stream, which receives an XML document $D$ of length $N$ as input and checks whether the result of $Q$ on $D$ is empty.

Considering the node-selecting XML query language *Core XPath* (cf., [14]), let $Q(D)$ denote the set of nodes that $Q$ selects in an XML document $D$. In [21], the following tight bounds on the complexity of processing *Core XPath* queries have been obtained:

(1) For every Core XPath query $Q$ there is an algorithm which decides for an input XML document $D$, whether $Q(D) \neq \emptyset$. This algorithm performs a single pass over its input and uses internal memory of size $O(h(D))$, where $h(D)$ denotes the height of the tree representation of $D$.

(2) There is a Core XPath query $Q$ such that for all functions $r, s$ with $r(H) \cdot s(H) = o(H)$, there exists no algorithm on one r/w stream which, when given as input an XML document $D$, decides whether $Q(D) \neq \emptyset$ and uses at most $r(h(D))$ head reversals and internal memory space of size at most $s(h(D))$.

The upper bound (1) is proved by standard automata theoretic techniques. For the lower bound (2), one can again use the lower bound for the *set disjointness* problem.

Let us end this subsection with an example exposing that *randomized* $(r, s, 1)$-bounded algorithms are computationally much stronger than deterministic ones. We introduce randomization to the computation model in such a way that in each computation step a coin may be tossed to decide what to do in this step. The probability $\Pr(A \text{ accepts } w)$ that input $w$ gets accepted by algorithm $A$ is then defined in a straightforward way (see [33, 20] for precise definitions).

Let us consider the problem *short-multiset-equality*, which receives as input two lists of bitstrings $x_1, \ldots, x_m$ and $y_1, \ldots, y_m$, where each $x_i$ and each $y_j$ has length at most $2 \log m$ (for arbitrary $m$). The task is to decide whether the multisets $\{x_1, \ldots, x_m\}$ and $\{y_1, \ldots, y_m\}$ are equal, i.e. whether they contain the same elements with the same multiplicities. Note that the input length is $N = \Theta(m \log m)$.

An easy communication complexity argument shows that there is no deterministic $(r, s, 1)$-bounded algorithm on one r/w stream that solves this problem with $r(N) \cdot s(N) = o(N)$. In particular, this implies that every deterministic algorithm that solves the problem *short-multiset-equality* by performing a single pass over the input requires internal memory of size $\Omega(N)$ (see [33] for details). However, by using standard fingerprinting techniques, the problem can be solved by a randomized algorithm with internal memory of size $O(\log N)$ as follows:

**Theorem 3.3 ([20])** *There is a randomized algorithm which, when given the parameter $m$, performs a single pass over the input and solves the short-multiset-equality problem with internal memory of size $O(\log N)$ such that*

- *each "yes"-instance is accepted with probability 1, and*
- *each "no"-instance is rejected with probability $\geqslant 2/3$.*

*Proof sketch:* The basic idea is to identify the input strings $x_i, y_j$ with natural numbers and to associate two polynomials $f(z) := \sum_{i=1}^{m} z^{x_i}$ and $g(z) := \sum_{j=1}^{m} z^{y_j}$ with the input $x_1, \ldots x_m, y_1 \ldots y_m$. Obviously, the two polynomials are equal if, and only if, the input is a "yes"-instance of the *short-multiset-equality* problem.

The basic idea of the algorithm is to choose a random number $r$, to evaluate $f(r)$ and $g(r)$, and to accept if, and only if, $f(r) = g(r)$. This algorithm will accept with probability 1 if the input is a "yes"-instance. However, if the input is a "no"-instance, the two polynomials $f$ and $g$ only have few common points, and thus the algorithm will reject with high probability.

A closer look shows that this procedure can be implemented by performing a single pass over the input. The algorithm computes $f(r)$ and $g(r)$ on-the-fly and uses arithmetic modulo a prime number of moderate size such that $O(\log N)$ bits of internal memory suffice. □

## 3.2 Several r/w streams

Let us now turn to the computation model where an arbitrary number $t$ of r/w streams are available.

A straightforward implementation of the *merge sort* algorithm shows that the problem *short-sorting* can be solved by using three r/w streams, $O(\log N)$ head reversals, and internal memory of size $O(\log N)$. I.e., *short-sorting* can be solved by an $(O(\log N), O(\log N), 3)$-algorithm on r/w streams. By using suitable coding tricks, one can perform sorting even with internal memory of size $O(1)$ and with a total number of two r/w streams [10]. Thus, using two r/w streams instead of a single one substantially increases the power of the computation model (cf., Theorem 3.2, stating that any $(r, s, 1)$-bounded algorithm for *short-sorting* requires that $r(N) \cdot s(N) = \Omega(N)$). The following theorem shows that this is essentially optimal: If instead of $O(\log N)$ only $o(\log N)$ head reversals are available, even internal memory of size $O(N^{1-\varepsilon})$ and an arbitrary number $t$ of r/w streams do not suffice to solve *short-sorting*.

**Theorem 3.4 ([20])** *Let $t$ be a positive integer and let $\varepsilon$ be an arbitrary number with $0 < \varepsilon < 1$.*
*The problem short-sorting cannot be solved by any $(o(\log N), O(N^{1-\varepsilon}), t)$-bounded algorithm on r/w streams.*

To prove this theorem, straightforward communication complexity based arguments (as used in the previous subsection) utterly fail. The reason is that we can easily communicate arbitrarily many bits from one part of the input to any other part by just copying the first part to a second r/w stream and then reading it in parallel with the second part of the input. This requires no internal memory and just two head reversals on the r/w streams.

But still, although the use of several r/w streams allows to copy large consecutive segments of the input from one place to another, there seems no easy way of significantly *permuting* the input without using too many head reversals. This intuition was confirmed in [22, 20]. Note that, unlike the lower bounds mentioned in Section 2, Theorem 3.4 does not rely on any kind of "indivisibility" assumption. Furthermore, the lower bound of Theorem 3.4 even holds for randomized algorithms which output either the correctly sorted sequence or the answer *"I don't know"*, and where the answer *"I don't know"* is returned with probability $\leqslant 1/2$ (cf., [20]).

Note that the *short-multiset-equality* problem as well as the *set disjointness* problem can easily be reduced to the sorting problem. Thus, both problems can be solved by a deterministic $(O(\log N), O(1), 2)$-bounded algorithm on r/w streams. Furthermore, from Theorem 3.3 we obtain a randomized $(O(1), O(\log N), 1)$-bounded algorithm for *short-multiset-equality* which never produces false negative answers, and which produces false positive answers with probability $\leqslant 1/3$. The next theorem shows that no corresponding algorithm with complementary acceptance and rejectance probabilities exists (not even when allowing up to $o(\log N)$ head reversals, internal memory of size $O(N^{1-\varepsilon})$, and an arbitrary number of r/w streams).

**Theorem 3.5 ([20])** *Let $t$ be a positive interger and let $\varepsilon$ be an arbitrary number with $0 < \varepsilon < 1$.*
*There is no $(o(\log N), O(N^{1-\varepsilon}), t)$-bounded randomized algorithm for the short-multiset-equality problem on r/w streams such that*

- *each "no"-instance is rejected with probability 1, and*
- *each "yes"-instance is accepted with probability $\geqslant 2/3$.*

For the *set disjointness* problem, in the presence of considerably less than $\log N$ head reversals, not even a randomized algorithm with 2-sided bounded error exists:

**Theorem 3.6 ([7])** *Let $t$ be a positive interger and let $\varepsilon$ be an arbitrary number with $0 < \varepsilon < 1$.*
*There is no $(o(\log N/\log\log N), O(N^{1-\varepsilon}), t)$-bounded randomized algorithm for the set disjointness problem on r/w streams such that*

- *each "no"-instance is rejected with probability $\geqslant 2/3$,*
- *each "yes"-instance is accepted with probability $\geqslant 2/3$.*

The above results easily lead to the following statements on the data complexity of relational algebra queries and queries posed against XML data [20, 7] (for any choice of $t \geqslant 1$ and $\varepsilon$ with $0 < \varepsilon < 1$).

(1) For every relational algebra query $Q$, the problem of evaluating $Q$ on a stream consisting of the tuples of the input database relations, can be solved by an $(O(\log N), O(1), 2)$-bounded deterministic algorithm on r/w streams.

(2) There exists a relational algebra query $Q_1$ such that the problem of evaluating $Q_1$ on a stream of the tuples of the input database relations cannot be solved by any $(o(\log N), O(N^{1-\varepsilon}), t)$-bounded algorithm on r/w streams.

(3) The task of checking whether the join of two relations $A$ and $B$ is empty cannot be performed by any $(o(\log N/\log\log N), O(N^{1-\varepsilon}), t)$-bounded randomized algorithm on r/w streams with a two-sided bounded error of at most 1/3.

(4) There is an XQuery query $Q_2$ such that the problem of evaluating $Q_2$ on an input XML document of length $N$ cannot be solved by any $(o(\log N), O(N^{1-\varepsilon}), t)$-bounded algorithm on r/w streams.

(5) There is an XPath query $Q_3$ such that the problem of checking, for an input XML document $D$ of length $N$, whether $Q_3(D) \neq \emptyset$, cannot be solved by any $(o(\log N), O(N^{1-\varepsilon}), t)$-bounded randomized algorithm on r/w streams with 1-sided bounded error that accepts "yes"-instances with probability 1 and that rejects "no"-instances with probability $\geqslant 2/3$.

(6) There is an XQuery query $Q_4$ such that the problem of checking whether the result of $Q_4$ on an input XML document of length $N$ is empty cannot be solved by any $(o(\log N/\log\log N), O(N^{1-\varepsilon}), t)$-bounded randomized algorithm on r/w streams with a two-sided bounded error of at most 1/3.

**Suggestions for further reading:** An overview of the model of r/w streams is given in [33]; technical details can be found in the articles [21, 20, 25, 7, 6].

A related computation model based on r/w streams and intermediate sorting steps is the *StrSort* model of [3] that was further considered in [32]. In [12], the *W-Stream* model, a restriction of the StrSort model in which intermediate sorting steps are prohibited, was introduced. This model can also be viewed as a restriction of the computation model of r/w streams with a single r/w stream, where only forward scans are allowed.

**An important future task:** It would be interesting to further study the computational power of the model with multiple r/w streams and intermediate sorting steps. First steps in this direction were taken in [3, 32]. Similarly as with the external memory model from Section 2, the lower bound proofs currently known for this model rely on the assumption that data items are indivisible and that only comparisons between data items are allowed. It is a challenging future task to develop methods for proving lower bounds in the StrSort model that do not rely on such an indivisibility assumption.

# 4 Finite Cursor Machines

Finite cursor machines (FCMs, for short) were introduced in [19] as an abstract model of database query processing. Informally, they can be described as follows:

The input for an FCM is a relational database. Each relation is represented by a *table*, i.e., an ordered list of rows, where each row corresponds to a tuple in the relation. The *size* $n$ of the input is defined as the sum of the number of rows of each input table.

Data elements are viewed as "indivisible" objects that can be manipulated by a number of built-in operations. We assume that all data items belong to a fixed, infinite universe $\mathbb{D}$, which is equipped with a number of built-in predicates (including, e.g., the equality predicate and a linear order). This feature is very convenient for modeling standard operations on data types like integers, floating point numbers, or strings, which may all be part of the set

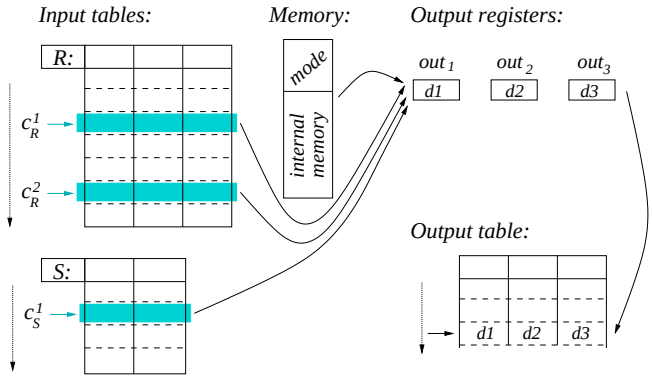



Figure 3: The model of finite cursor machines.

$\mathbb{D}$. The universe $\mathbb{D}$ together with the built-in predicates form the so-called *background structure*.

FCMs can operate in a finite number of *modes* (corresponding to states of a finite automaton). Additionally, they can use *internal memory* in which they can store bitstrings. We speak of $O(1)$-FCMs (respectively, $o(n)$-FCMs) when restricting attention to FCMs where internal memory is of constant size (respectively, of size $o(n)$). FCMs access each input relation through a finite number of *cursors*, each of which can read one row of a table at any time. The model incorporates certain *streaming* aspects by imposing a restriction on the movement of the cursors: They can move on the tables sequentially, in one direction only. Thus, once the last cursor has passed a row of a table, this row can never be accessed again during the computation. Note, however, that several cursors can be moved asynchronously over the same table at the same time, and thus, entries in different, possibly far apart, regions of the table can be read and processed simultaneously.

For producing an output table, a finite number of *output registers* are available, each of them capable of storing an element in $\mathbb{D}$. Whenever the so-called *outputmode* is set to a special value *out*, the machine automatically produces the next output tuple: The tuple consisting of the values in the output registers (in some predefined order) is appended to the output table.

An illustration of the model is given in Figure 3. The formal notion of FCMs is defined in the framework of *abstract state machines*; details can be found in [19].

Figure 4 gives an example of an FCM program that computes the query $Q$ defined on a ternary relation $R$ over $\mathbb{N}$ that returns the sum of the first and second attribute of each row whose third attribute contains a value bigger than 100. In the example, there is a single output register $out_1$ and a single cursor $c$ that accesses the rows of $R$, starting in the first row. The program starts with outputmode set to the value *init* and is applied again and again until the cursor has been moved beyond the last row of $R$. The command $c := next_R(c)$ advances the

```
 1: if outputmode = out then
 2:   par
 3:     outputmode := init
 4:     c := next_R(c)
 5:   endpar
 6: else
 7:   if attribute^3_R(c) > 100 then
 8:     par
 9:       out_1 := attribute^1_R(c) + attribute^2_R(c)
10:       outputmode := out
11:     endpar
12:   else
13:     c := next_R(c)
14:   endif
15: endif
```

Figure 4: Example of an FCM program.

cursor to the next row of $R$. The par/endpar construct encloses lines of code that are executed in parallel. The code in lines 7–11 enforces that if the third column of the currently visited row contains a value $> 100$, then the output register $out_1$ is filled with the sum of the values in the row's first two columns, and the outputmode *out* is activated, enforcing that the content of register $out_1$ is appended as a new tuple of the output table. In the next execution of the program, the code in lines 1–5 deactivates the outputmode (by setting it to the value *init*) and advances the cursor to the next tuple in $R$.

FCMs model quite faithfully what happens in relational database query processing. For example, the *selection* $\sigma_\theta(R)$ of all rows of $R$ that satisfy a certain selection condition $\theta$ can be implemented by an FCM in a straightforward way. If tables are *sorted*, then also the *projection* operator and the *union*, *intersection*, and *difference* of two relations can be accomplished by an FCM. Also the *semijoin*[1] $R \ltimes_\theta S$ can be computed by an FCM operating on sorted tables, provided that the join condition $\theta$ consists of a conjunction of equalities and at most two inequalities [19]. This leads to the following observation concerning the *semijoin algebra*, i.e., the restriction of relational algebra where, instead of joins, only semijoins are allowed.

**Theorem 4.1 ([19])** *Every semijoin algebra query can be computed by a query plan composed of a finite number of $O(1)$-FCMs and sorting operations.*

Notice that FCMs are capable of computing some relational algebra queries that are not expressible in the semijoin algebra. An example is the Boolean query asking whether $R = \pi_1(R) \times \pi_2(R)$, i.e., asking whether the binary relation $R$ is the cartesian product of the projection of $R$ to its first and second component, respectively. Also, examples of queries that are computable by FCMs but not expressible in relational algebra are known (cf., [19]).

Furthermore, *sliding window joins* for a fixed window size $w$ (enforcing that the join operator is successively applied to portions of the data, each portion consisting of a number $w$ of consecutive rows of an input table) can be easily computed by an FCM. Note, however, that general *joins* cannot be computed since the output size of a join may be quadratic in the size of the input, while $O(1)$-FCMs can output only a linear number of tuples.

In connection with Theorem 4.1, the question arises whether intermediate sorting steps are really necessary. The next theorem answers this question affirmatively. Let $R$, $S$, and $T$ be relations, such that $S$ has arity $\geqslant 2$, and consider the composition

$$R \ltimes_{x_1=y_1} (S \ltimes_{x_2=y_1} T)$$

of two semijoins. $S \ltimes_{x_2=y_1} T$ returns all tuples $s$ in $S$ for which there exists a tuple $t$ in $T$ whose first component coincides with the second component of $s$. $R \ltimes_{x_1=y_1} (S \ltimes_{x_2=y_1} T)$ returns all tuples $r$ in $R$ for which there exists a tuple $s$ in $(S \ltimes_{x_2=y_1} T)$ whose first component coincides with the first component of $r$.

**Theorem 4.2 ([19])**
*The query "Is $R \ltimes_{x_1=y_1} (S \ltimes_{x_2=y_1} T)$ nonempty?", where $R$ and $T$ are unary relations and $S$ is a binary relation, is not computable by any $o(n)$-FCM, even if the input consists of all sorted versions of the relations $R$, $S$, $T$.*

Here, the notion of "sorted versions" of a relation is defined as follows. Recall that the universe $\mathbb{D}$ of data items is equipped with a linear order $<$. We consider sorting in ascending or descending order. Then, a relation of arity $p$ can be sorted lexicographically in $p! \cdot 2^p$ different ways: For any permutation $\pi$ of $\{1, \ldots, p\}$ and any *ordering scheme* $\sigma : \{1, \ldots, p\} \to \{\text{asc}, \text{desc}\}$, let $sort_{\pi,\sigma}$ be the operation that takes a $p$-ary relation $R$ and lexicographically sorts it as follows: It sorts the $\pi(1)$-th column first (ascendingly or descendingly, depending on the value of $\sigma(1)$), the $\pi(2)$-th column second (ascendingly or descendingly, depending on the value of $\sigma(2)$), ..., and sorts the $\pi(p)$-th column last (ascendingly or descendingly, depending on the value of $\sigma(p)$). Theorem 4.2 assumes for any relation, that for any applicable permuation $\pi$ and ordering scheme $\sigma$, the table sorted according to $sort_{\pi,\sigma}$ is present as an input table.

Note that Theorem 4.2 is sharp in terms of arity of the input relations and in terms of size of internal memory: If $S$ is unary (and $R$ and $T$ of arbitrary arities), then the corresponding query is computable by an $O(1)$-FCM on sorted inputs. Furthermore, if internal memory of size $O(n)$ (rather than $o(n)$) is available, then the entire input database can be loaded into internal memory, and the query can be processed there.

[1] $R \ltimes_\theta S$ returns all tuples $r$ of $R$ for which there exists a tuple $s$ of $S$ such that $(r, s)$ satisfies condition $\theta$.

**Suggestions for further reading:** The formal definition of FCMs as well as detailed proofs of the results mentioned in this section can be found in [19].

**An important future task:** The main open question from [19] is as follows: Is there a Boolean relational algebra query that cannot be computed by a finite composition of FCMs and sorting operations? The conjectured answer is "yes", since every query that can be computed by a finite composition of sorting operations and $O(1)$-FCMs (over an efficiently decidable background structure) can be evaluated by a deterministic algorithm the performs $O(n \log n)$ steps (where $n$ denotes the size of the input database) — and reasonable conjectures in parameterized complexity (cf., [18]) imply that there are Boolean relational algebra queries that cannot be evaluated in time $O(n \log n)$ (with respect to data complexity).

# 5 Mpms-Automata

In XML databases, an *index* over an XML document typically consists of a number of *streams*, one for each label that occurs in the document. We write $T_{\mathtt{a}}$ to denote the stream associated with label `a`. Each stream $T_{\mathtt{a}}$ consists of *positional encodings* of all elements, in document order, that occur in the document and that carry the label `a`.

A widely used encoding scheme is the *BEL encoding* (cf., e.g., [8, 36]), in which each element is encoded as a triple $(Begin, End, Level)$, where *Begin* and *End* denote the positions of the starting tag and ending tag of the element in the document, and *Level* denotes the level at which the corresponding node occurs in the associated XML document tree. An example of an XML document, the BEL encoding of its elements, and the corresponding index streams is given in Figure 5.

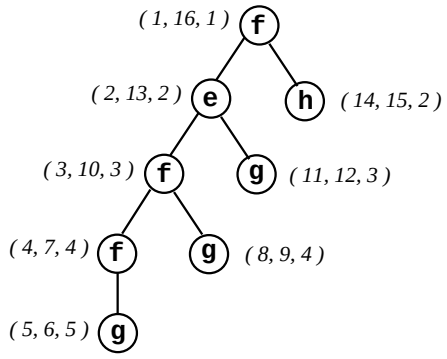
Most currently known algorithms for processing so-called *twig join queries* (i.e., *XPath* queries of a particular kind) can be viewed as implementations of the following computation model (cf., e.g., [8, 36]). Let us consider a fixed query $Q$ — for example, the query `//e/g` asking for all `g`-labeled nodes whose parent carries label `e`. The input to the evaluation algorithm is an indexed XML document, i.e., a collection of streams $T_{\mathtt{a}}$ of positional encodings, for each label `a` occurring in the document. For each occurrence of each label `a` in the *query*, the algorithm may use a *cursor* (or, *head*), with which the stream $T_{\mathtt{a}}$ can be processed once from left to right. An algorithm can move heads asynchronously, and it can read from a head position many times, until it decides to advance it to the next position. The output is written to a write-only output stream.

In [36], Shalem and Bar-Yossef gave a precise characterization of the memory requirement for evaluating twig queries with this computation model. Among other results, they showed the following.

XML doc.:

```
1:  <f>
2:   <e>
3:    <f>
4:     <f>
5:      <g>
6:      </g>
7:     </f>
8:     <g>
9:     </g>
10:   </f>
11:   <g>
12:   </g>
13:  </e>
14:  <h>
15:  </h>
16: </f>
```

XML tree with BEL encoding:




Corresponding index streams:

| | |
|---|---|
| $T_{\mathtt{e}}$ : $(2, 13, 2)$ | $T_{\mathtt{f}}$ : $(1, 16, 1),\ (3, 10, 3),\ (4, 7, 4)$ |
| $T_{\mathtt{h}}$ : $(14, 15, 2)$ | $T_{\mathtt{g}}$ : $(5, 6, 5),\ (8, 9, 4),\ (11, 12, 3)$ |

Figure 5: XML document, tree with BEL encoding, and corresponding index streams.

**Theorem 5.1 ([36])** *Let* `e` *and* `g` *be two distinct labels, and let $\mathcal{A}$ be an algorithm in the above computation model that answers the question "Is there an* `e`*-labeled node that has a* `g`*-labeled child" on any indexed XML input document. Then, for every $n \geqslant 1$ there is an XML document $D$ of depth $n$ on which $\mathcal{A}$ uses at least $n - O(\log n)$ bits of memory.*

For reasoning about the power of the above computation model, the notion of *mpms-automata*[2], introduced in [34], is useful. An illustration of the model is given in Figure 6. The formal definition of the model is as follows.

Let $\mathbb{D}$ be a (potentially infinite) set of data items, let $t, m, k_f, k_b$ be integers with $t, m \geqslant 1$ and $k_f, k_b \geqslant 0$. An

*mpms-automaton $\mathcal{A}$ with parameters $(t, \mathbb{D}, m, k_f, k_b)$*

receives as input $t$ streams $S_1, \ldots, S_t$ of elements in $\mathbb{D}$ (formally, we can view each stream $S_i$ as a finite string over alphabet $\mathbb{D}$, i.e., as an element in $\mathbb{D}^*$).

The automaton's memory consists of $m$ different states (note that this corresponds to a memory buffer consisting of $\log m$ bits). The automaton's state space is denoted by $Q$. We assume that $Q$ contains a designated *start state* and that there is a designated subset $F$ of $Q$ of *accepting states*. On each of the input streams, the automaton has $k_f$ heads that process the stream from left to right (so-called *forward heads*) and $k_b$ heads that process the stream from right to left (so-called *backward heads*). The heads are allowed to move asynchronously. We use $k$ to denote the total number of heads, i.e., $k = tk_f + tk_b$. In the *initial configuration* of $\mathcal{A}$ on input $S_1, \ldots, S_t$ the automaton is in the start state, all forward heads are

[2] *mpms* is short for multi-pass processing of multiple streams

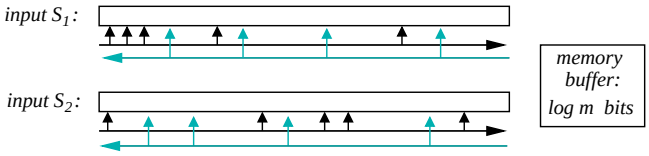



Figure 6: An mpms-automaton with $t = 2$ input streams, $k_f = 5$ forward heads, and $k_b = 4$ backward heads.

placed on the leftmost element, and all backward heads are placed on the rightmost element in the corresponding stream.

During each computation step, depending on (a) the current state (i.e., the current content of the automaton's memory) and (b) the elements of $S_1, \ldots, S_t$ at the current head positions, a transition function determines (1) the next state (i.e., the new content of the automaton's memory), (2) which of the $k$ heads is advanced to the next position (where forward heads are advanced one step to the right, and backward heads are advanced one step to the left), and which (sequence of) elements of $\mathbb{D}$ is written to the output stream.

The automaton's computation on input $S_1, \ldots, S_t$ ends as soon as each head has passed the entire stream. The input is *accepted* if the automaton's state then belongs to the set $F$ of accepting states, and it is *rejected* otherwise.

Consider, e.g., the variant $Disj_n$ of the *set disjointness* problem with input domain $\mathbb{D}_n := \{a_i, b_i, c_i : i \in \{1, \ldots, n\}\}$, where the input consists of two streams $S_1 = s_1 s_2 \cdots s_n$ and $S_2 = t_1 t_2 \cdots t_n$. The goal is to decide whether the sets $\{s_1, \ldots, s_n\}$ and $\{t_1, \ldots, t_n\}$ are disjoint. It is not difficult to see that the problem $Disj_n$ can be solved by an mpms-automaton with parameters $(2, \mathbb{D}_n, n+2, \sqrt{n}, 0)$ (cf., [34]). An according *lower bound* is given by the following theorem.

**Theorem 5.2 ([34])** *For all $n$, $m$, $k_f$, $k_b$ such that, for $k = 2k_f + 2k_b$ and $v = (k_f^2 + k_b^2 + 1) \cdot (2k_f k_b + 1)$,*

$k^2 \cdot v \cdot \log(n+1) + k \cdot v \cdot \log m + v \cdot (1 + \log v) \leqslant n$,

*the problem $Disj_n$ cannot be solved by any mpms-automaton with parameters $(2, \mathbb{D}_n, m, k_f, k_b)$.*

In [36], the variant $RDisj_n$ of the problem $Disj_n$ was considered, where inputs are restricted to streams $S_1 = s_1 \cdots s_n$ and $S_2 = t_1 \cdots t_n$ where, for each $i \in \{1, \ldots, n\}$, $s_i \in \{a_i, b_i\}$ and $t_{n-i+1} \in \{a_i, c_i\}$. Theorem 5.1 can then be proved by (1) using the lower bound for $RDisj_n$ stated in the next theorem and (2) reducing the $RDisj_n$ problem to the problem of answering the query stated in Theorem 5.1.

**Theorem 5.3 (implicit in [36])** *The problem $RDisj_n$ cannot be solved by any mpms-automaton with parameters $(2, \mathbb{D}_n, m, 1, 0)$ for $m < \frac{2^n}{2n}$.*

**Suggestions for further reading:** Details on the computation model for processing indexed XML documents can be found in [8, 36] and the references given therein. The formal definition of the model of mpms-automata is given in [34].

**An important future task:** Consider also randomized versions of mpms-automata, design efficient randomized approximation algorithms for particular problems, and develop techniques for proving lower bounds in the randomized model.

# 6 Stream Processing

The *data stream* scenario considers data that is not stored but, instead, has to be processed on-the-fly by using only a limited amount of memory. Thus, the basic data stream model can be viewed as the restriction of the computational model of read/write streams where a single forward scan can be performed on a single r/w stream (cf., Section 3.1). When designing data stream algorithms, one aims at algorithms whose memory size is far smaller than the size of the input.

Typical application areas for which data stream (or, *one-pass*) processing is relevant are, e.g., IP network traffic analysis, mining text message streams, or processing meteorological data generated by sensor networks. Data stream algorithms are also used to support query optimization in relational database systems. In fact, virtually all query optimization methods in relational database systems rely on information about the number of distinct values of an attribute or the self-join size of a relation — and these pieces of information have to be maintained while the database is updated. Algorithms for accomplishing this task have been introduced in the seminal paper [4].

*Lower bounds* on the size of memory needed for solving a problem by a one-pass algorithm are usually obtained by applying methods from *communication complexity* (cf., Section 3.1). In fact, for many concrete problems it is known that the memory needed for solving the problem by a deterministic one-pass algorithm is at least linear in the size $N$ of the input. For some of these problems, however, *randomized* one-pass algorithms can still compute good *approximate* answers while using memory of size sublinear in $N$. Typically, such algorithms are based on *sampling*, i.e., only a "representative" portion of the data is taken into account, and *random projections*, i.e., only a rough "sketch" of the data is stored in memory. An example of such an algorithm is given in the proof of Theorem 3.3.

It should be noted that many of the sophisticated data stream algorithms achieve a surprisingly good performance (cf., [28]). For example, it is not at all obvious how to maintain information on the number of *distinct* elements that occurred in a stream, without storing a list of all those elements (since, intuitively, for each element that arrives, one has to check whether this is a *new* ele-

ment or just the repetition of an element that had already occurred in the stream). Here, randomized algorithms are known which give good approximate solutions to this problem while using just a logarithmic number of bits [4].

**Suggestions for further reading.** In recent years, the database community has addressed the issue of designing general-purpose *data stream management systems* and query languages that are suitable for new application areas where massive amounts of transient data have to be processed. To get an overview of this research area, [5] is a good starting point. Foundations for a theory of stream queries have been laid in [23]. A comprehensive overview of efficient algorithms for data stream processing can be found in [28]. In the context of XML query processing and validation, stream-based approaches have been examined in detail (cf., e.g., [35, 17, 9, 30]).

# 7 MapReduce & the Mud Model

Implementations of the *MapReduce* programming model, e.g., by Google [11] or the Apache Hadoop project [24], are comonnly used frameworks for distributed processing of large datasets. Users give a high-level specification of an algorithm in terms of a *map* and a *reduce* function. The underlying system then automatically parallelizes the computation across large-scale clusters of machines, handles failures, and schedules inter-machine communication to make efficient use of the network and the disks.

In this programming model, the input consists of a set of (key, value) pairs. The *map* function takes an input pair and produces a set of intermediate (key, value) pairs. The system then automatically groups together all intermediate values with the same intermediate key $k$ and passes them to the *reduce* function. The *reduce* function merges (or, *aggregates*) these values together to form a smaller set of values for that key (typically, just a single value is produced for each key $k$).

In [13], Feldman et al. introduced an abstract algorithmic model for massive, unordered, distributed (mud, for short) computation, as implemented by systems for MapReduce. For the special case that just a single key is available, a *mud algorithm* is a triple $A = (\Phi, \oplus, \eta)$ with the following parameters:

- $\Phi : \mathbb{D} \to Q$, where $\mathbb{D}$ is the domain of potential input data items, and $Q$ is the set of intermediate values (or, *messages*),
- $\oplus : Q \times Q \to Q$ maps two messages to a single message,
- $\eta : Q \to \mathbb{D}$ is the so-called *post-processing* operator that produces the final output.

When the algorithm is executed on an input $X = x_1, \ldots, x_n$ with $x_i \in \mathbb{D}$, the function $\Phi$ produces the sequence of messages $Y = y_1, \ldots, y_n = \Phi(x_1), \ldots, \Phi(x_n)$. The operator $\oplus$ is used to aggregate these messages into a single message. Note that the output can depend on the order in which $\oplus$ is applied. Formally, for any binary tree $\mathcal{T}$ with $n$ leaves, and for any permutation $\pi$ of $\{1, \ldots, n\}$, let $m_{\mathcal{T},\pi}(X)$ denote the message $q \in Q$ that results from applying $\oplus$ along the topology of $\mathcal{T}$ with the sequence $y_{\pi(1)}, \ldots, y_{\pi(n)}$ used as input at the leaves of $\mathcal{T}$. The overall output of the mud algorithm then is the data item $A(X) := \eta(m_{\mathcal{T},\pi}(X))$. It should be emphasized that $\mathcal{T}$ and $\pi$ are *not* part of the algorithm, but rather, the algorithm designer needs to make sure that $\eta(m_{\mathcal{T},\pi}(X))$ is independent of the particular choice of $\mathcal{T}$ and $\pi$. This is required to ensure that mud algorithms serve as an abstract model of *distributed* computations that are independent of the underlying implementation. A sufficient condition that guarantees independence of $\mathcal{T}$ and $\pi$ is associativity and commutativity of $\oplus$.

We say that a function $f : \mathbb{D}^n \to \mathbb{D}$ is *computed* by a mud algorithm $A$ if $f(X) = A(X)$ for all $X \in \mathbb{D}^n$. Obviously, mud algorithms can only compute *symmetric* functions, i.e., functions $f$ where $f(x_1, \ldots, x_n) = f(x_{\pi(1)}, \ldots, x_{\pi(n)})$ for any permutation $\pi$.

The *communication complexity* of a mud algorithm is defined as $\log |Q|$, i.e., the number of bits needed to represent a message. The *space* (resp., *time*) *complexity* of a mud algorithm is defined as the maximum space (resp., time) complexity of its component functions $\Phi$, $\oplus$, $\eta$. Any mud algorithm can be simulated by a data stream algorithm in a straightforward way. A main technical result of [13] shows that, in some sense, also the reverse is true:

**Theorem 7.1 ([13])** *Any deterministic streaming algorithm that computes a symmetric function $f : \mathbb{D}^n \to \mathbb{D}$ can be simulated by a mud algorithm with the same communication complexity, and the square of its space complexity.*

**Suggestions for further reading:** The mud model and some extensions were introduced in [13]. An overview of MapReduce can be found in [11, 24].

**An important future task:** The *time* complexity of the simulation provided by Theorem 7.1 is superpolynomial, and thus the simulation does not immediately provide distributed algorithms of high performance. It would be interesting to develop more time-efficient simulations.

**Acknowledgments.** I thank Monika Henzinger and S. Muthu Muthukrishnan for sending me valuable information on the mud model. Furthermore, I thank Katharina Hahn for helpful comments on an earlier version of this paper.

# References

[1] J. Abello and J. Vitter, editors. *External Memory Algorithms*, volume 50. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1999.

[2] A. Aggarwal and J. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

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

[4] 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.

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

[6] P. Beame and D.-T. Huynh-Ngoc. On the value of multiple read/write streams for approximating frequency moments. In *Proc. FOCS'08*, pages 499–508, 2008.

[7] P. Beame, T. Jayram, and A. Rudra. Lower bounds for randomized read/write stream algorithms. In *Proc. STOC'07*, pages 689–698, 2007.

[8] N. Bruno, N. Koudas, and D. Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proc. SIGMOD'02*, pages 310–321, 2002.

[9] C. Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient filtering of XML documents with XPath expressions. *VLDB Journal*, 11(4):354–379, 2002.

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

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[12] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proc. SODA'06*, pages 714–723, 2006.

[13] J. Feldman, S. Muthukrishnan, A. Sidiropoulos, C. Stein, and Z. Svitkina. On distributing symmetric streaming computations. In *Proc. SODA'08*, pages 710–719, 2008.

[14] G. Gottlob, C. Koch, and R. Pichler. The complexity of XPath query evaluation. In *Proc. PODS'03*, pages 179–190, 2003.

[15] G. Gottlob, N. Leone, and F. Scarcello. The complexity of acyclic conjunctive queries. *Journal of the ACM*, 48(3):431–498, 2001.

[16] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[17] T. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu. Processing XML streams with deterministic automata and stream indexes. *ACM Transactions on Database Systems*, 29(4):752–788, 2004.

[18] M. Grohe. Parameterized complexity for the database theorist. *SIGMOD Record*, 31(4):86–96, 2002.

[19] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. Van den Bussche. Database query processing using finite cursor machines. *Theory of Computing Systems*, 44(4):533–560, 2009.

[20] M. Grohe, A. Hernich, and N. Schweikardt. Lower bounds for processing data with few random accesses to external memory. *Journal of the ACM*, 56(3):1–58, 2009.

[21] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. *Theor. Comput. Sci.*, 380(1-2):199–217, 2007.

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

[23] Y. Gurevich, D. Leinders, and J. Van den Bussche. A theory of stream queries. In *Proc. DBPL'07*, pages 153–168, 2007.

[24] Hadoop: Open-source software for reliable, scalable, distributed computing. `http://hadoop.apache.org/`.

[25] A. Hernich and N. Schweikardt. Reversal complexity revisited. *Theor. Comput. Sci.*, 401(1-3):191–205, 2008.

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

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

[28] S. Muthukrishnan. Data Streams: Algorithms and Applications. *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[29] M. Nodine and J. Vitter. Greed sort: An optimal sorting algorithm for multiple disks. *Journal of the ACM*, 42(4):919–933, 1995.

[30] D. Olteanu. SPEX: Streamed and progressive evaluation of XPath. *IEEE Trans. Knowl. Data Eng.*, 19(7):934–949, 2007.

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

[32] M. Ruhl. *Efficient Algorithms for New Computational Models*. PhD thesis, MIT, 2003.

[33] N. Schweikardt. Machine models and lower bounds for query processing. In *Proc. PODS'07*, pages 41–52, 2007.

[34] N. Schweikardt. Lower bounds for multi-pass processing of multiple data streams. In *Proc. STACS'09*, pages 51–61, 2009.

[35] L. Segoufin and C. Sirangelo. Constant-memory validation of streaming XML documents against DTDs. In *Proc. ICDT'07*, pages 299–313, 2007.

[36] M. Shalem and Z. Bar-Yossef. The space complexity of processing XML twig queries over indexed documents. In *Proc. ICDE'08*, pages 824–832, 2008. Full version available at `http://webee.technion.ac.il/people/zivby/`.

[37] L. J. Stockmeyer. *The complexity of decision problems in automata and logic*. PhD thesis, MIT, 1974.

[38] M. Y. Vardi. The complexity of relational query languages. In *Proc. STOC'82*, pages 137–146, 1982.

[39] J. Vitter. External memory algorithms. In *Proc. PODS'98*, pages 119–128, 1998.

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

[41] J. Vitter and E. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2–3):110–147, 1994.

[42] M. Yannakakis. Algorithms for acyclic database schemes. In *Proc. VLDB'81*, pages 82–94, 1981.