Database Query Processing Using Finite Cursor Machines

Martin Grohe[1] Yuri Gurevich[2] Dirk Leinders[3]
Nicole Schweikardt[1] Jerzy Tyszkiewicz[4]
Jan Van den Bussche[3]

[1]Humboldt-University Berlin
[2]Microsoft Research
[3]Hasselt University and Transnational University of Limburg
[4]University of Warsaw

Abstract

We introduce a new abstract model of database query processing, *finite cursor machines*, that incorporates certain data streaming aspects. The model describes quite faithfully what happens in so-called "one-pass" and "two-pass query processing". Technically, the model is described in the framework of abstract state machines. Our main results are upper and lower bounds for processing relational algebra queries in this model, specifically, queries of the semijoin fragment of the relational algebra.

1 Introduction.

We introduce and analyze *finite cursor machines*, an abstract model of database query processing. Data elements are viewed as "indivisible" abstract objects with a vocabulary of arbitrary, but fixed, functions. Relational databases consist of finitely many finite relations over the data elements. Relations are considered as tables whose rows are the tuples in the relation. Finite cursor machines can operate in a finite number of *modes* using an *internal memory* in which they can store bit strings. They access each relation through finitely many cursors, each of which can read one row of a table at any time. The answer to a query, which is also a relation, can be given through a suitable output mechanism. The model incorporates certain "streaming" or "sequential processing" aspects by imposing two restrictions: First, the cursors can only move on the tables sequentially in one direction. Thus once the last cursor has left a row of a table, this row can never be accessed again during the computation. Second, the internal memory is limited. For our lower bounds, it will be sufficient to put an $o(n)$ restriction on the internal memory size, where n is the size (that is, the number of entries) of the input database. For the upper bounds, no internal memory will be needed. The model is clearly inspired by the *abstract state machine (ASM)* methodology [16], and indeed we will formally define our model using this methodology. The model was first presented in a talk at the ASM 2004 workshop [29].

Algorithms and lower bounds in various data stream models have received considerable attention in recent years both in the theory community (e.g., [1, 2, 5, 6, 13, 14, 18, 25]) and the database systems community (e.g., [3, 4, 7, 12, 15, 20, 26]). Note that our model is fairly powerful; for example, the multiple cursors can easily be used to perform multiple sequential scans of the input data. But more than that; by moving several cursors asynchronously over the same table, entries in different, possibly far apart, regions of the table can be read and processed simultaneously. This way, different regions of the same or of different tables can "communicate" with each other without requiring any internal memory, which makes it difficult to use communication complexity to establish lower bounds. The model is also powerful in that it allows arbitrary functions to access and process data elements. This feature is very convenient to model "built in" standard operations on data types like integers, floating point numbers, or strings, which may all be part of the universe of data elements.

Despite these powerful features, the model is weak in many respects. We show that a finite cursor machine with internal memory size $o(n)$ cannot even test whether two sets A and B, given as lists, are disjoint, even if besides the lists A and B, also their reversals are given as input. However, if two sets A and B are given as *sorted* lists, a machine can easily compute the intersection. Aggarwal et al. [1] have already made a convincing case for combining streaming computations with sorting, and we will consider an extension of the model with a sorting primitive.

Our main results are concerned with evaluating *relational algebra queries* in the finite cursor machine model. Relational algebra forms the core of the standard query language SQL and is thus of fundamental importance for databases. We prove that, when all sorted versions of the database relations are provided as input, every operator of the relational algebra can be computed, except for the *join*. The latter exception, however, is only because the output size of a join can be quadratic, while finite cursor machines by their very definition can output only a linear number of different tuples. A *semijoin* is a projection of a join between two relations to the columns of one of the two relations (note that the projection prevents the result of a semijoin from getting larger than the relations to which the semijoin operation is applied). The *semijoin algebra* is then a natural fragment of the relational algebra that may be viewed as a generalization of acyclic conjunctive queries [9, 22, 21, 30]. When sorted versions of the database relations are provided as input, semijoins can be computed by finite cursor machines. Consequently, every query in the semijoin fragment of the relational algebra can be computed by a composition of finite cursor machines and sorting operations. This is interesting because it models quite faithfully what is called "one-pass" and "two-pass processing" in database systems [11]. The question then arises: are intermediate sorting operations really needed? Equivalently, can every semijoin-algebra query already be computed by a single machine on sorted inputs? We answer this question negatively in a very strong way, and this is our main technical result: Just a composition of two semijoins $R \ltimes (S \ltimes T)$ with R and T unary relations and S a binary relation is not computable by a finite cursor machine with internal memory size $o(n)$ working on sorted inputs. This result is quite sharp, as we will indicate.

The paper is structured as follows: After fixing some notation in Section 2, the notion of finite cursor machines is introduced in Section 3. The power of $O(1)$-FCMs and of $o(n)$-FCMs is investigated in Sections 4 and 5. Some concluding remarks and open questions can be found in Section 6.

2 Preliminaries.

Throughout the paper we fix an arbitrary, typically infinite, universe $\mathbb{E}$ of "data elements", and we fix a database schema $\mathcal{S}$. I.e., $\mathcal{S}$ is a finite set of relation names, where each relation name has an associated arity, which is a natural number. A database $\mathbf{D}$ with schema $\mathcal{S}$ assigns to each $R \in \mathcal{S}$ a finite, nonempty set $\mathbf{D}(R)$ of k-tuples of data elements, where k is the arity of R. In database terminology the tuples are often called *rows*. The *size* of database $\mathbf{D}$ is defined as the total number of rows in $\mathbf{D}$.

A *query* is a mapping Q from databases to relations, such that the relation $Q(\mathbf{D})$ is the answer of the query Q to database $\mathbf{D}$. The *relational algebra* is a basic language used in database theory to express exactly those queries that can be composed from the actual database relations by applying a sequence of the following operations: union, intersection, difference, projection, selection, and join. The meaning of the first three operations should be clear, the *projection* operator $\pi_{i_1,\ldots,i_k}(R)$ returns the projection of a relation R to its components $i_1, \ldots, i_k$, the *selection* operator $\sigma_{p(i_1,\ldots,i_k)}(R)$ returns those tuples from R whose i_1th, $\ldots$, i_kth components satisfy the predicate p, and the *join* operator $R \bowtie_\theta S$ (where θ is a conjunction of equalities of the form $\bigwedge_{s=1}^{k} x_{i_s} = y_{j_s}$) is defined as $\{(\overline{a}, \overline{b}) : \overline{a} \in R,\ \overline{b} \in S,\ a_{i_s} = b_{j_s}$ for all $s \in \{1, \ldots, k\}\}$. A natural sub-language of the relational algebra is the so-called *semijoin algebra* where, instead of ordinary joins, only *semijoin* operations of the form $R \ltimes_\theta S$ are allowed, defined as $\{\overline{a} \in R : \exists \overline{b} \in S : a_{i_s} = b_{j_s}$ for all $s \in \{1, \ldots, k\}\}$.

To formally introduce our computation model, we need some basic notions from mathematical logic such as (many-sorted) vocabularies, structures, terms, and atomic formulas.

3 Finite Cursor Machines.

In this section we formally define *finite cursor machines* using the methodology of Abstract State Machines (ASMs). Intuitively, an ASM can be thought of as a transition system whose states are described by many-sorted first-order structures (or algebras)[1]. Transitions change the interpretation of some of the symbols—those in the *dynamic* part of the vocabulary—and leave the remaining symbols—those in the *static* part of the vocabulary—unchanged. Transitions are described by a finite collection of simple update rules, which are "fired" simultaneously (if they are inconsistent, no update is carried out). A crucial property of the sequential ASM model, which we consider here, is that in each transition only a limited part of the state is changed. The de-

[1]Beware that "state" refers here to what for Turing machines is typically called "configuration"; the term "mode" is used for what for Turing machines is typically called "state".

tailed definition of sequential ASMs is given in the Lipari guide [16], but our presentation will be largely self-contained.
We now describe the formal model of finite cursor machines.

The Vocabulary: The *static vocabulary* of a finite cursor machine (FCM) consists of two parts, Υ_0 (providing the background structure) and $\Upsilon_{\mathcal{S}}$ (providing the particular input).

Υ_0 consists of three sorts: $\mathsf{Element}$, $\mathsf{Bitstring}$, and Mode. Furthermore, Υ_0 may contain an arbitrary number of functions and predicates, as long as the output sort of each function is $\mathsf{Bitstring}$. Finally, Υ_0 contains an arbitrary but finite number of constant symbols of sort Mode, called *modes.* The modes *init*, *accept*, and *reject* are always in Υ_0.

$\Upsilon_{\mathcal{S}}$ provides the input. For each relation name $R \in \mathcal{S}$, there is a sort Row_R in $\Upsilon_{\mathcal{S}}$. Moreover, if the arity of R is k, we have function symbols $\mathit{attribute}_R^i\colon \mathsf{Row}_R \to \mathsf{Element}$ for $i = 1, \dots, k$. Furthermore, we have a constant symbol $\perp_R$ of sort Row_R. Finally, we have a function symbol $\mathit{next}_R\colon \mathsf{Row}_R \to \mathsf{Row}_R$ in $\Upsilon_{\mathcal{S}}$.

The *dynamic vocabulary* Υ_M of an FCM M contains only constant symbols. This vocabulary always contains the symbol *mode* of sort Mode. Furthermore, there can be a finite number of symbols of sort $\mathsf{Bitstring}$, called *registers.* Moreover, for each relation name R in the database schema, there are a finite number of symbols of sort Row_R, called *cursors on R.*

The Initial State: Our intention is that FCMs will work on databases. Database relations, however, are sets, while FCMs expect lists of tuples as inputs. Therefore, formally, the input to a machine is an *enumeration* of a database, which consists of enumerations of the database relations, where an enumeration of a relation is simply a listing of all tuples in some order. An FCM M that is set to run on an enumeration of a database $\mathbf{D}$ then starts with the following structure $\mathcal{M}$ over the vocabulary $\Upsilon_0 \cup \Upsilon_{\mathcal{S}} \cup \Upsilon_M$: The interpretation of $\mathsf{Element}$ is $\mathbb{E}$; the interpretation of $\mathsf{Bitstring}$ is the set of all finite bitstrings; and the interpretation of Mode is simply given by the set of modes themselves. For technical reasons, we must assume that $\mathbb{E}$ contains an element $\perp$. For each $R \in \mathcal{S}$, the sort Row_R is interpreted by the set $\mathbf{D}(R) \cup \{\perp_R\}$; the function $\mathit{attribute}_R^i$ is defined by $(x_1, \dots, x_k) \mapsto x_i$, and $\perp_R \mapsto \perp$; finally, the function next_R maps each row to its successor in the list, and maps the last row to $\perp_R$. The dynamic symbol *mode* initially is interpreted by the constant *init*; every register contains the empty bitstring; and every cursor on a relation R contains the first row of R.

The Program of an FCM: A *program* for the machine M is now a program as defined as a basic sequential program in the sense of ASM theory, with the important restriction that all basic updates concerning a cursor c on R must be of the form $c := next_R(c)$.

Thus, basic update rules of the following three forms are rules: $mode := t$, $r := t$, and $c := next_R(c)$, where t is a term over $\Upsilon_0 \cup \Upsilon_{\mathcal{S}} \cup \Upsilon_M$, and r is a register and c is a cursor on R. The semantics of these rules is the obvious one: Update the dynamic constant by the value of the term. Update rules $r_1, \ldots, r_m$ can be combined to a new rule $\mathsf{par}\ r_1\ \ldots\ r_m\ \mathsf{endpar}$, the semantics of which is: Fire rules $r_1, \ldots, r_m$ in parallel; if they are inconsistent do nothing. Furthermore, if r_1 and r_2 are rules and φ is an atomic formula over $\Upsilon_0 \cup \Upsilon_{\mathcal{S}} \cup \Upsilon_M$, then also $\mathsf{if}\ \varphi\ \mathsf{then}\ r_1\ \mathsf{else}\ r_2\ \mathsf{endif}$ is a rule. The semantics is obvious.

Now, an FCM program is just a single rule. (Since finitely many rules can be combined to one using the $\mathsf{par} \ldots \mathsf{end}$ construction, one rule is enough.)

The Computation of an FCM: Starting with the initial state, successively apply the (single rule of the FCM's) program until *mode* is equal to *accept* or to *reject*. Accordingly, we say that M terminates and *accepts*, respectively, *rejects* its input.

Given that inputs are *enumerations* of databases, we must be careful to define the result of a computation on a database. We agree that an FCM *accepts* a database $\mathbf{D}$ if it accepts *every* enumeration of $\mathbf{D}$. This already allows us to use FCMs to compute decision queries. In the next paragraph we will see how FCMs can output lists of tuples. We then say that an FCM M computes a query Q if on each database $\mathbf{D}$, the output of M on *any* enumeration of $\mathbf{D}$ is an enumeration of the relation $Q(\mathbf{D})$. Note that later we will also consider FCMs working only on sorted versions of database relations: in that case there is no ambiguity.

Producing Output: We can extend the basic model so that the machine can output a list of tuples. To this end, we expand the dynamic vocabulary Υ_M with a finite number of constant symbols of sort $\mathsf{Element}$, called *output registers*, and with a constant of sort Mode, called the *output mode*. We expand the static vocabulary Υ_0 with a number of functions with output sort Element, called *output functions.* These output functions can only be used to update the output registers. The output registers can be updated following the normal rules of ASMs. The output registers, however, can not be used as an argument to a static function.

In each state of the finite cursor machine, when the output mode is equal to the special value *out*, the tuple consisting of the values in the output registers (in some predefined order) is output; when the output mode is different from *out*, no tuple is output. In the initial state each output register contains the value $\perp$ and the output mode is equal to *init*. We denote the output of a machine M working on a database $\mathbf{D}$ by $M(\mathbf{D})$.

Space Restrictions: For considering FCMs whose bitstring registers are restricted in size, we use the following notation: Let M be a finite cursor machine and $\mathcal{F}$ a class of functions from $\mathbb{N}$ to $\mathbb{N}$. Then we say that M is an $\mathcal{F}$*-machine* (or, an $\mathcal{F}$*-FCM*) if there is a function $f \in \mathcal{F}$ such that, on each database enumeration $\mathbf{D}$ of size n, the machine only stores bitstrings of length $f(n)$ in its registers. We are mostly interested in $O(1)$-FCMs and $o(n)$-FCMs. Note that the latter are quite powerful. For example, such machines can easily store the positions of the cursors. On the other hand, $O(1)$-machines are equivalent to FCMs that do not use registers at all (because bitstrings of constant length could also be simulated by finitely many *modes*).

Example 3.1. Consider a query Q defined on a ternary relation $R(A, B, C)$ over $\mathbb{N}$ that returns the sum of the A- and B-attributes for each row with a C-attribute at least 100. Let $\mathbb{E}$ be the set of natural numbers $\mathbb{N}$. Consider a static vocabulary containing at least the predicate "> 100" and the output function $+$ on $\mathbb{N}$. Then an FCM can compute query Q with a single cursor and a single output register. The following FCM program computes Q.

```
if $outputmode = out$ then
    par
        $outputmode := init$
        $c := next_R(c)$
    endpar
else
    if $attribute^3_R(c) > 100$ then
        par
            $outputmode := out$
            $out_1 := attribute^1_R(c) + attribute^2_R(c)$
        endpar
    else
        $c := next_R(c)$
    endif
endif
```

3.1 Discussion of the Model.

Storing Bitstrings instead of Data Elements: An important question about our model is the strict separation between data elements and bitstrings. Indeed, data elements are abstract entities, and our background structure may contain arbitrary functions and predicates, mixing data elements and bitstrings, with the important restriction that the output of a function is always a bitstring. At first sight, a simpler way to arrive at our model would be without bitstrings, simply considering an arbitrary structure on the universe of data elements. Let us call this variation of our model the "universal model".

Note that the universal model can easily become computationally complete. It suffices that finite strings of data elements can somehow be represented by other data elements, and that the background structure supplies the necessary manipulation functions for that purpose. Simple examples are the natural numbers with standard arithmetic, or the strings over some finite alphabet with concatenation. Thus, if we would want to prove complexity lower bounds in the universal model, while retaining the abstract nature of data elements and operations on them, it would be necessary to formulate certain logical restrictions on the available functions and predicates on the data elements. Finding interesting such restrictions is not clear to us. In the model with bitstrings, however, one can simply impose restrictions on the length of the bitstrings stored in registers, and that is precisely what we will do. Of course, the unlimited model with bitstrings can also be computationally complete. It suffices that the background structure provides a coding of data elements by bitstrings.

Element Registers: The above discussion notwithstanding, it might still be interesting to allow for registers that can remember certain data elements that have been seen by the cursors, but without arbitrary operations on them. Formally, we would expand the dynamic vocabulary Υ_M with a finite number of constant symbols of sort Element, called *element registers*. It is easy to see, however, that such element registers can already be simulated by using additional cursors, and thus do not add anything to the basic model.

Running Time and Output Size: A crucial property of FCMs is that all cursors are one-way. In particular, an FCM can perform only a linear number of steps where a cursor is advanced. As a consequence, an $O(1)$-FCM with output can output only a linear number of different tuples. On the other hand, if the background structure is not restricted in any way, arbitrary computations on the register contents can occur in between cursor advancements. As a matter of fact, in this paper we will present a number of positive results and a number of negative results. For the positive results, registers will never be needed, and in particular, FCMs run in linear time. For the negative results, arbitrary computations on the registers will be allowed.

Look-ahead: Note that the terms in the program of an FCM can contain nested applications of the function $next_R$, such as $next_R(next_R(c))$. In some sense, such nestings of depth up to d correspond to a *look-ahead* where the machine can access the current cursor position as well as the next d positions. It is, however, straightforward to see that every k-cursor FCM with look-ahead $\leq d$ can be simulated by a $(k\times d)$-cursor FCM with look-ahead 0. Thus, throughout the remainder of this paper we will w.l.o.g. restrict attention to FCMs that have look-ahead 0, i.e., to FCMs where the function $next_R$ never occurs in if-conditions or in update rules of the form $mode := t$ or $r := t$.

The Number of Cursors: In principle we could allow more than constantly many cursors, which would enable us to store that many data elements. We stick with the constant version for the sake of technical simplicity, and also because our *upper* bounds only need a constant number of cursors. Note, however, that our main *lower* bound result can be extended to a fairly big number of cursors (cf. Remark 5.3).

4 The Power of $O(1)$-Machines.

We start with a few simple observations on the database query processing capabilities of FCMs, with or without sorting, and show that sorting is really needed.

Let us first consider *compositions* of FCMs in the sense that one machine works on the outputs of several machines working on a common database.

Proposition 4.1. *Let $M_1, \dots, M_r$ be FCMs working on a schema $\mathcal{S}$, let $\mathcal{S}'$ be the output schema consisting of the names and arities of the output lists of $M_1, \dots, M_r$, and let M_0 be an FCM working on schema $\mathcal{S}'$. Then there exists an FCM M working on schema $\mathcal{S}$, such that $M(\mathbf{D}) = M_0(\mathbf{D}')$, for each database $\mathbf{D}$ with schema $\mathcal{S}$ and the database $\mathbf{D}'$ that consists of the output relations $M_1(\mathbf{D}), \dots, M_r(\mathbf{D})$.*

The proof is obvious: Each row in a relation R_i of database $\mathbf{D}'$ is an output row of a machine M_i working on $\mathbf{D}$. Therefore, each time M_0 moves a cursor on R_i, the desired finite cursor machine M will simulate that part of the computation of M_i on $\mathbf{D}$ until M_i outputs a next row.

Let us now consider the operators from relational algebra: Clearly, *selection* can be implemented by an $O(1)$-FCM. Also, *projection* and *union* can easily be accomplished if either duplicate elimination is abandoned or the input is given in a suitable order. *Joins*, however, are *not* computable by an FCM, simply because the output size of a join can be quadratic, while $O(1)$-FCMs can output only a linear number of different tuples.

In stream data management research [4], one often restricts attention to *sliding window joins* for a fixed window size w. This means that the join operator is successively applied to portions of the data, each portion consisting of a number w of consecutive rows of the input relations. The following example illustrates how an $O(1)$-FCM can compute a sliding window join.

Example 4.2. Consider a sliding window join of $R(A, B)$ and $S(C, D)$ with condition $B = C$ where the windows slide simultaneously on either relation by the size of the windows, say w (on both R and S). A finite cursor machine for this job has w cursors c_R^i on R, and w cursors c_S^i on S, for $i = 1, \dots, w$. The machine begins by advancing the ith cursor $i - 1$ times on each of the two relations. Then, all pairs of cursors are considered, and joining tuples are output, using rules of the following form for $1 \le i, j \le w$:

```
if $mode = check_{i,j}$ and $attribute_R^1(c_R^i) = attribute_S^1(c_S^j)$ then
  par
    $outputmode := out$
    $out_1 := attribute_R^1(c_R^i)$
    $out_2 := attribute_R^2(c_R^i)$
    $out_3 := attribute_S^1(c_S^j)$
```

```
        $out_4 := attribute^2_S(c^j_S)$
        $mode := next\text{-}mode$
    endpar
endif
```

Next, all cursors are advanced w times. This continues until the end of the relations. This machine has a large number of similar rules, which could be automatically generated or executed from a high-level description.

Of course, the general case with relations of arbitrary arity, and arbitrary join condition θ can be treated in the same way.

Using more elaborate methods, we can moreover show that even checking whether the join is nonempty (so that output size is not an issue) is hard for FCMs. Specifically, we will consider the problem whether two sets intersect, which is the simplest kind of join. We will give two proofs: an elegant one for $O(1)$-machines, using a proof technique that is simple to apply, and an intricate one for more general $o(n)$-machines (Theorem 5.4). Note that the following result is valid for *arbitrary* (but fixed) background structures.

Theorem 4.3. *There is no $O(1)$-FCM that checks for two sets R and S whether $R \cap S \neq \emptyset$.*

Proof. Let M be an $O(1)$-FCM that is supposed to check whether $R \cap S \neq \emptyset$. Without loss of generality, we assume that $\mathbb{E}$ is totally ordered by a predicate $<$ in Υ_0. Using Ramsey's theorem, we can find an infinite set $V \subseteq \mathbb{E}$ over which the truth of the atomic formulas in M's program on tuples of data elements only depends on the way these data elements compare w.r.t. $<$ (details on this can be found, e.g., in Libkin's textbook [24, Section 13.3]). Now choose $2n$ elements in V, for n large enough, satisfying $a_1 < a'_1 < \cdots < a_n < a'_n$, and consider the run of M on $R = \{a_1, \ldots, a_n\}$ (listed in that order) and $S = \{a'_n, \ldots, a'_1\}$. We say that a pair of cursors "checks" i if in some state during the run, one of the cursors is on a_i and the other one is on a'_i. By the way the lists are ordered, every pair of cursors can check only one i. Hence, some j is not checked. Now replace a'_j in S by a_j. The machine will not notice this, because a_j and a'_j have the same relative order with respect to the other elements in the lists. The intersection of R and S, however, is now nonempty, so M is wrong. □

Of course, when the sets R and S are given as *sorted* lists, an FCM can easily compute $R \cap S$ by performing one simultaneous scan over the two lists. Moreover, while the full join is still not computable simply because its output is too large, the semijoin $R \ltimes S$ is also easily computed by an FCM on sorted inputs. Furthermore, the same holds for the difference $R - S$. These easy observations motivate us to extend FCMs with sorting, in the spirit of "two-pass query processing" based on sorting [11].

Formally, assume that $\mathbb{E}$ is totally ordered by a predicate $<$ in Υ_0. Then a relation of arity p can be sorted "lexicographically" in $p!$ different ways: for any permutation ρ of $\{1, \ldots, p\}$, let sort_ρ denote the operation that sorts a p-ary relation $\rho(1)$-th column first, $\rho(2)$-th column second, and $\rho(p)$-th column last. By an FCM *working on sorted inputs* of a database $\mathbf{D}$, we mean an FCM that gets all possible sorted orders of all relations of $\mathbf{D}$ as input lists. We then summarize the above discussion as follows:

Proposition 4.4. *Each operator of the semijoin algebra (i.e, union, intersection, difference, projection, selection, and semijoin) can be computed by an $O(1)$-FCM on sorted inputs.*

Corollary 4.5. *Every semijoin algebra query can be computed by a composition of $O(1)$-FCMs and sorting operations.*

Proof. Starting from the given semijoin algebra expression we replace each operator by a composition of one FCM with the required sorting operations. □

The simple proof of the above corollary introduces a lot of intermediate sorting operations. In some cases, intermediate sorting can be avoided by choosing in the beginning a particularly suitable ordering that can be used by *all* the operations in the expression [28].

Example 4.6. Consider the query $(R - S) \ltimes_{x_2 = y_2} T$, where R, S and T are binary relations. Since the semijoin compares the second columns, it needs its inputs sorted on second columns first. Hence, if $R - S$ is computed on $\mathsf{sort}_{(2,1)}(R)$ and $\mathsf{sort}_{(2,1)}(S)$ by some machine M, then the output of M can be piped directly to a machine M' that computes the semijoin on that output and on $\mathsf{sort}_{(2,1)}(T)$. By compositionality (Proposition 4.1), we can then even compose M and M' into a single FCM. A stupid way to compute the same query would be to compute $R - S$ on $\mathsf{sort}_{(1,2)}(R)$ and $\mathsf{sort}_{(1,2)}(S)$, thus requiring a re-sorting of the output.

The question then arises: can intermediate sorting operations always be avoided? Equivalently, can every semijoin algebra query already be computed by a single machine on sorted inputs? We can answer this negatively. Our proof applies a known result from the classical topic of multihead automata, which is indeed to be expected given the similarity between multihead automata and FCMs.

Specifically, the *monochromatic 2-cycle* query about a binary relation E and a unary relation C asks whether the directed graph formed by the edges in E consists of a disjoint union of 2-cycles where the two nodes on each cycle either both belong to C or both do not belong to C. Note that this query is indeed expressible in the semijoin algebra as *"Is $e_1 \cup e_2 \cup e_3$ empty?"*, where $e_1 := E - (E \underset{\substack{x_2=y_1\\x_1=y_2}}{\ltimes} E)$, where $e_2 := E \underset{\substack{x_2=y_1\\x_1\neq y_2}}{\ltimes} E$, and where $e_3 := (E \underset{x_1=y_1}{\ltimes} C) \underset{x_2=y_1}{\ltimes} ((\pi_1(E) \cup \pi_2(E)) - C)$

(We use a nonequality in the semijoin condition, but that is easily incorporated in our formalism as well as computed by an FCM on sorted inputs.)

Before proving that the monochromatic 2-cycle query can not be computed by an $O(1)$-FCM on sorted inputs, we recall the result on multihead automata as a lemma.

One-way multihead deterministic finite state automata are devices with a finite state control, a single read-only tape with a right endmarker \$ and a finite number of reading heads which move on the tape from left to right. Computation on an input word w starts in a designated state q_0 with all reading heads adjusted on the first symbol of w. Depending on the internal state and the symbols read by the heads, the automaton changes state and moves zero or more heads to the right. An input word w is accepted if a final state is reached when all heads are adjusted on the endmarker \$. A one-way multihead deterministic finite state automaton with k heads is denoted by 1DFA(k). A one-way multihead deterministic *sensing* finite state automaton, denoted by 1DSeFA(k), is a 1DFA(k) that has the ability to detect when heads are on the same position. Formal definitions have been given by Rosenberg [27].

For natural numbers n and f, consider the following formal languages over the alphabet $\{a, b\}$:

$$L_n^f := \{w_1bw_2b\cdots bw_fbw'_fb\cdots bw'_2bw'_1 \mid \forall i = 1,\ldots,f : w_i, w'_i \in \{a,b\}^* \text{ and } |w_i| = |w'_i| = n\}$$

$$P_n^f := \{w_1bw_2b\cdots bw_fbw'_fb\cdots bw'_2bw'_1 \in L_n^f \mid \forall i = 1,\ldots,f : w_i^R = w'_i\}$$

Lemma 4.7 (Hromkovič [19]). *Let M be a one-way, k-head, sensing DFA, and let $f > \binom{k}{2}$. Then for sufficiently large n, if M accepts all strings in P_n^f, then M also accepts a string in $L_n^f - P_n^f$.*

Proof. On any string in P_n^f, consider the pattern of "prominent" configurations of M, where a prominent configuration is a halting one, or one in which a head has just left a w_i or a w_i' and is now on a b. If s is the number of internal states of the automaton, there are at most $s \cdot (2f(n+1))^k$ different configurations, of which at most $2fk$ are prominent, so there are at most

$$p(n) := \big(s \cdot (2f(n+1))^k\big)^{2fk}$$

different such patterns. As there are 2^{fn} different strings in P_n^f, there is a group G of at least $2^{fn}/p(n)$ different strings in P_n^f with the same pattern.[2]

On any $w_1bw_2b\ldots bw_fbw_f^Rb\ldots bw_2^Rbw_1^R \in P_n^f$, we say that M "checks" region $i \in \{1,\ldots,f\}$ if at some point during the run, there is a head in w_i, and another head in w_i^R. Every pair of heads can check at most one i, so since $f > \binom{k}{2}$, at least one i is not checked.

In our group G, the non-checked i is the same for all strings, because they have the same pattern. If we group the strings in G further on their parts outside w_i and w_i^R, there are at most $2^{(f-1)n}$ different groups, so there is a subgroup H of G of at least $2^n/p(n)$ different strings that agree outside w_i and w_i^R. For sufficiently large n, we have $2^n/p(n) \geq 2$.

We have arrived at two strings in P_n^f of the form

$$y_1 = w_1bw_2b..bw_ib..bw_nbw_n^Rb..bw_i^Rb..bw_2^Rbw_1^R$$
$$y_2 = w_1bw_2b..bw_i'b..bw_nbw_n^Rb..bw_i'^Rb..bw_2^Rbw_1^R$$

with $w_i \neq w_i'$, and with the same pattern. But then M will also accept the following string $y \in L_n^f - P_n^f$:

$$w_1bw_2b\cdots bw_ib\cdots bw_nbw_n^Rb\cdots bw_i'^Rb\cdots bw_2^Rbw_1^R$$

Indeed, while M is in w_i, no head is in w_i^R and thus the run behaves as on y_1; while M is in $w_i'^R$, no head is in w_i and thus the run behaves as on y_2. Since y_1 and y_2 have the same pattern, y has that pattern as well and hence y is accepted. □

[2]We do not use the word "group" in the mathematical sense but in its sense as a normal English word.

We are now able to prove:

Theorem 4.8. *The monochromatic 2-cycle query is not computable by an $O(1)$-FCM on sorted inputs.*

Proof. The proof is via a reduction from the Palindrome problem. Note that Lemma 4.7 still holds if we equip a 1DSeFA(k) with an arbitrary but finite number of oblivious right-to-left heads that can only move from right to left on the input tape sensing other heads, but not read the symbols on the tape. Prominent configurations and checking a region i are then defined in terms of the normal, non-oblivious left-to-right heads. As a corollary we have that there is no 1DSeFA(k) with oblivious right-to-left heads that recognizes the language $P := \{w \in \{0,1\}^* \mid w = w^{\mathcal{R}}\}$ of palindromes.

Now let M be an $O(1)$-FCM that is supposed to solve the monochromatic 2-cycle query. Again using Ramsey's theorem, we can find an infinite set $V \subseteq \mathbb{E}$ over which the truth of the atomic formulas in M's program on tuples of data elements only depends on the way these data elements compare w.r.t. $<$ (see Theorem 4.3). Hence, there is an $O(1)$-FCM M' with only $<$ in its rules that is equivalent to M over V. We now come to the reduction. Given a string $w = w_1 \cdots w_n$ over $\{0,1\}$, we choose n values $a_1 < \cdots < a_n \in V$. Then define relation E as $\{(a_i, a_{n-i+1}) \mid 1 \leq i \leq n\}$ and define relation C as $\{a_i \mid w_i = 1\}$. It is clear that w is a palindrome if and only if E and C form a positive instance to the monochromatic 2-cycle query. From FCM M' we can construct a 1DSeFA(k) with oblivious right-to-left heads that would recognize P as follows:

- each cursor on E corresponds to a pair consisting of a "normal" left-to-right head and an oblivious right-to-left head;
- each cursor on C corresponds to a normal head;
- the internal state of the automaton keeps track of the mode of the finite cursor machine, together with the relative order of the elements seen by all cursors;
- each time a cursor on E is advanced, the normal head of the corresponding pair of heads is moved to the right and the oblivious head is moved to the left, while sensing makes sure that the internal state of the automaton is changed according to the new relative order;

- each time a cursor on C is advanced, the corresponding head is moved to the next 1 on the input tape.

We conclude that FCM M can not exist. $\square$

An important remark is that the above proof only works if the set C is only given in ascending order. In practice, however, one might as well consider sorting operations in descending order, or, for relations of higher arity, arbitrary mixes of ascending and descending orders on different columns. Indeed, that is the general format of sorting operations in the database language SQL. We thus extend our scope to sorting in descending order, and to much more powerful $o(n)$-machines, in the next section.

5 Descending Orders and the Power of $o(n)$-Machines.

We already know that the computation of semijoin algebra queries by FCMs and sortings in ascending order only requires intermediate sortings. So, the next question is whether the use of descending orders can avoid intermediate sorting. We will answer this question negatively, and will do this even for $o(n)$-machines (whereas Theorem 4.8 is proven only for $O(1)$-machines).

Formally, on a p-ary relation, we now have sorting operations $\mathsf{sort}_{\rho,f}$, where ρ is as before, and $f\colon \{1,\dots,p\} \to \{\nearrow,\searrow\}$ indicates ascending or descending. To distinguish from the terminology of the previous section, we talk about an FCM working on *AD-sorted inputs* to make clear that both ascending and descending orders are available.

Before we show our main technical result, we remark that the availability of sorted inputs using descending order allows $O(1)$-machines to compute more relational algebra queries. Indeed, we can extract such a query from the proof of Theorem 4.8. Specifically, the "Palindrome" query about a binary relation R and a unary relation C asks whether R is of the form $\{(a_i, a_{n-i+1}) \mid i = 1,\dots,n\}$ with $a_1 < \cdots < a_n$, and $C \subseteq \{a_1,\dots,a_n\}$ such that $a_i \in C \Leftrightarrow a_{n-i+1} \in C$. We can express this query in the relational algebra (using the order predicate in selections). In the following proposition, the lower bound was already shown in Theorem 4.8, and the upper bound is easy.

Proposition 5.1. *The "Palindrome" query cannot be solved by an $O(1)$-FCM on sorted inputs, but can be solved by an $O(1)$-FCM on AD-sorted inputs.*

We now establish:

Theorem 5.2. *The query RST := "Is $R \ltimes_{x_1=y_1} (S \ltimes_{x_2=y_1} T)$ nonempty?", where R and T are unary and S is binary, is not computable by any $o(n)$-FCM working on AD-sorted inputs.*

Proof. For the sake of contradiction, suppose M is a $o(n)$-FCM computing RST on sorted inputs. Without loss of generality, we can assume that M accepts or rejects the input only when all cursors are positioned at the end of their lists.

Let k be the total number of cursors of M, let r be the number of registers and let m be the number of modes occurring in M's program. Let $v := \binom{k}{2}+1$.

Choose n to be a multiple of v^2, and choose $4n$ values in $\mathbb{E}$ satisfying $a_1 < a'_1 < a_2 < a'_2 < \cdots < a_n < a'_n < b_1 < b'_1 < \cdots < b_n < b'_n$.

Divide the ordered set $\{1,\ldots,n\}$ evenly in v consecutive blocks, denoted by $B_1,\ldots,B_v$. So, B_i equals the set $\{(i-1)\frac{n}{v}+1,\ldots,i\frac{n}{v}\}$. Consider the following permutation of $\{1,\ldots,n\}$:

$$\pi : \quad (i-1)\cdot\tfrac{n}{v}+s \ \mapsto\ (v-i)\cdot\tfrac{n}{v}+s$$

for $1 \le i \le v$ and $1 \le s \le \frac{n}{v}$. So, π maps subset B_i to subset B_{v-i+1}, and vice versa.

We fix the binary relation S of size $2n$ for the rest of this proof as follows:

$$S \ := \ \big\{(a_\ell, b_{\pi\ell}) : \ \ell \in \{1,..,n\}\big\} \cup \big\{(a'_\ell, b'_{\pi\ell}) : \ \ell \in \{1,..,n\}\big\}.$$

Furthermore, for all sets $I, J \subseteq \{1,\ldots,n\}$, we define unary relations $R(I)$ and $T(J)$ of size n as follows:

$$\begin{aligned} R(I) &:= \{a_\ell : \ \ell \in I\} \ \cup \ \{a'_\ell : \ \ell \in I^c\} \\ T(J) &:= \{b_\ell : \ \ell \in J\} \ \cup \ \{b'_\ell : \ \ell \in J^c\}, \end{aligned}$$

where I^c denotes $\{1,\ldots,n\} - I$. By $\mathbf{D}(I,J)$, we denote the database consisting of the lists $\mathsf{sort}_{\nearrow}\big(R(I)\big)$, $\mathsf{sort}_{\searrow}\big(R(I)\big)$, $\mathsf{sort}_{\nearrow}\big(T(J)\big)$, $\mathsf{sort}_{\searrow}\big(T(J)\big)$, and all sorted versions of S. It is easy to see that the nested semijoin of $R(I)$, S, and $T(J)$ is empty if, and only if, $(\pi(I) \cap J) \cup (\pi(I)^c \cap J^c) = \emptyset$. Therefore, for each I, the query RST returns *false* on instance $\mathbf{D}(I, \pi(I)^c)$, which we will denote by $\mathbf{D}(I)$ for short. Furthermore, we observe for later use:

the query RST on $\mathbf{D}(I, \pi(J)^c)$ returns *true* if and only if $I \neq J$. $(*)$

To simplify notation a bit, we will in the following use $R_{\nearrow}$ and $T_{\nearrow}$ to denote lists $\mathsf{sort}_{\nearrow}\big(R(I)\big)$ and $\mathsf{sort}_{\nearrow}\big(T(I)\big)$ sorted in ascending order, and we use $R_{\searrow}$ and $T_{\searrow}$ to denote the lists $\mathsf{sort}_{\searrow}\big(R(I)\big)$ and $\mathsf{sort}_{\searrow}\big(T(I)\big)$ sorted in descending order.

Consider a cursor c on list $R_{\nearrow}$ of the machine M. In a certain state (i.e., configuration), we say that c is on position ℓ on $R_{\nearrow}$ if M has executed $\ell-1$ update rules $c := next_{R_{\nearrow}}(c)$. I.e., if cursor c is on position ℓ on $R_{\nearrow}$, then c sees value a_ℓ or a'_ℓ. We use analogous notation for the sorted lists $R_{\searrow}$, $T_{\nearrow}$, and $T_{\searrow}$. I.e., if a cursor c is on position ℓ on $R_{\searrow}$ (resp. $T_{\nearrow}$, resp. $T_{\searrow}$), then c sees value $a_{n-\ell+1}$ or $a'_{n-\ell+1}$ (resp. b_ℓ or b'_ℓ, resp. $b_{n-\ell+1}$ or $b'_{n-\ell+1}$).

Consider the run of M on $\mathbf{D}(I)$. We say that a pair of cursors of M *checks block* B_i if at some state during the run

- one cursor in the pair is on a position in B_i on $R_{\nearrow}$ (i.e., the cursor reads an element a_ℓ or a'_ℓ, for some $\ell \in B_i$) and the other cursor in the pair is on a position in B_{v-i+1} on $T_{\nearrow}$ (i.e., the cursor reads an element $b_{\pi\ell}$ or $b'_{\pi\ell}$, for some $\ell \in B_i$), or

- one cursor in the pair is on a position in B_{v-i+1} on $R_{\searrow}$ (i.e., the cursor reads an element a_ℓ or a'_ℓ, for some $\ell \in B_i$) and the other cursor in the pair is on a position in B_i on $T_{\searrow}$ (i.e., the cursor reads an element $b_{\pi\ell}$ or $b'_{\pi\ell}$, for some $\ell \in B_i$).

Note that each pair of cursors working on the ascendingly sorted lists $R_{\nearrow}$ and $T_{\nearrow}$ or on the descendingly sorted lists $R_{\searrow}$ and $T_{\searrow}$, can check at most one block. There are v blocks and at most $\binom{k}{2} < v$ cursor pairs. Hence, there is one block B_{i_0} that is not checked by any pair of cursors working on $R_{\nearrow}$ and $T_{\nearrow}$ or on $R_{\searrow}$ and $T_{\searrow}$. In order to also deal with pairs of cursors on $R_{\nearrow}$ and $T_{\searrow}$ or on $R_{\searrow}$ and $T_{\nearrow}$, we further divide each block B_i evenly into v consecutive subblocks, denoted by $B_i^1, \ldots, B_i^v$. So, B_i^j equals the set $\{(i-1)\frac{n}{v} + (j-1)\frac{n}{v^2} + 1, \ldots, (i-1)\frac{n}{v} + j\frac{n}{v^2}\}$. We say that a pair of cursors of M *checks subblock* B_i^j if at some state during the run

- one cursor in the pair is on a position in B_i^j on $R_{\nearrow}$ (thus reading an element a_ℓ or a'_ℓ, for some $\ell \in B_i^j$) and the other cursor in the pair is on a position in B_i^{v-j+1} on $T_{\searrow}$ (thus reading an element $b_{\pi\ell}$ or $b'_{\pi\ell}$, for some $\ell \in B_i^j$), or

- one cursor in the pair is on a position in B_{v-i+1}^{v-j+1} on $R_{\searrow}$ (thus reading an element a_ℓ or a'_ℓ, for some $\ell \in B_i^j$) and the other cursor in the pair is on a position in B_{v-i+1}^{j} on $T_{\nearrow}$ (thus reading an element $b_{\pi\ell}$ or $b'_{\pi\ell}$, for some $\ell \in B_i^j$).

Note that each pair of cursors working either on $R_{\nearrow}$ and $T_{\searrow}$ or on $R_{\searrow}$ and $T_{\nearrow}$, can check at most one subblock in B_{i_0}. There are v subblocks in B_{i_0} and at most $\binom{k}{2} < v$ cursor pairs. Hence, there is at least one subblock $B_{i_0}^{j_0}$ that is not checked by any pair of cursors working either on $R_{\nearrow}$ and $T_{\searrow}$ or on $R_{\searrow}$ and $T_{\nearrow}$. Note that, since the entire block B_{i_0} is not checked by any pair or cursors working either on $R_{\nearrow}$ and $T_{\nearrow}$ or on $R_{\searrow}$ and $T_{\searrow}$, the subblock $B_{i_0}^{j_0}$ is thus not checked by *any* pair of cursors (on $R_{\nearrow}$, $R_{\searrow}$, $T_{\nearrow}$, $T_{\searrow}$).

We say that M checks subblock B_i^j if at least one pair of cursors of M checks subblock B_i^j.

At this point it is useful to introduce the following terminology. By "block $B_{i_0}^{j_0}$ on R", we refer to the positions in $B_{i_0}^{j_0}$ of list $R_{\nearrow}$ and to the positions in $B_{v-i_0+1}^{v-j_0+1}$ of list $R_{\searrow}$, i.e., "block $B_{i_0}^{j_0}$ on R" contains values a_ℓ or a'_ℓ where $\ell \in B_{i_0}^{j_0}$. By "block $B_{i_0}^{j_0}$ on T", however, we refer to the positions in $B_{v-i_0+1}^{j_0}$ of list $T_{\nearrow}$ and to the positions in $B_{i_0}^{v-j_0+1}$ of list $T_{\searrow}$, i.e., "block $B_{i_0}^{j_0}$ on T" contains values $b_{\pi\ell}$ where $\ell \in B_{i_0}^{j_0}$. Note that this terminology is consistent with the way we have defined the notion of "checking a block".

Consider then the set $\mathcal{I}$ of 2^n instances, defined via $\mathcal{I} := \{\mathbf{D}(I) : I \subseteq \{1, \ldots, n\}\}$. We argued before that on each instance in $\mathcal{I}$, there is at least one subblock B_i^j that M does not check. Because there are only v^2 such possible subblocks and 2^n different instances in $\mathcal{I}$, there exists a set $\mathcal{I}_0 \subseteq \mathcal{I}$ of cardinality at least $2^n/v^2$ and 2 indices i_0 and j_0, such that M does not check subblock $B_{i_0}^{j_0}$ on any instance in $\mathcal{I}_0$.

Now we apply an averaging argument to fix all input elements outside the critical block $B_{i_0}^{j_0}$: We divide $\mathcal{I}_0$ into equivalence classes induced by the following equivalence relation:

$$\mathbf{D}(I) \equiv \mathbf{D}(J) \quad \Leftrightarrow \quad I - B_{i_0}^{j_0} = J - B_{i_0}^{j_0}$$

Since $B_{i_0}^{j_0}$ has $\frac{n}{v^2}$ elements, there are at most $2^{n-\frac{n}{v^2}}$ equivalence classes. Thus, since $\mathcal{I}_0$ has at least $2^n/v^2$ elements, there exists an equivalence class $\mathcal{I}_1 \subseteq \mathcal{I}_0$ of cardinality at least $\frac{2^n/v^2}{2^{n-\frac{n}{v^2}}} = 2^{\frac{n}{v^2}}/v^2$, such that for any $\mathbf{D}(I)$ and

$\mathbf{D}(J)$ in $\mathcal{I}_1$, we have $I - B_{i_0}^{j_0} = J - B_{i_0}^{j_0}$. Note that for larger and larger n, $2^{\frac{n}{v^2}}/v^2$ becomes arbitrarily large.

Let $\mathbf{D}(I)$ be an element of $\mathcal{I}_1$. Consider the run of M on $\mathbf{D}(I)$. Let c be a cursor and let $\mathcal{M}_c^I$ be the state of M in the run on $\mathbf{D}(I)$ when cursor c has just left block $B_{i_0}^{j_0}$ on R or on T. Let $\overline{\mathcal{M}^I}$ be the k-tuple consisting of these states $\mathcal{M}_c^I$ for all cursors c. This tuple $\overline{\mathcal{M}^I}$ can have only $2^{k \log m + k^2 \log 2n + k \cdot r \cdot o(n)}$ different values. To see this note that a state of the machine is completely determined by the machine's current *mode* (one out of m possible values), the positions of each of the k cursors (where each cursor can be in one out of at most $2n$ possible positions), and the contents of the r bitstring registers (each of which has length $o(n)$). Hence, there are only $m \cdot (2n)^k \cdot 2^{r \cdot o(n)}$ different states for M.

Since $\mathcal{I}_1$ has at least $2^{\frac{n}{v^2}}/v^2$ elements, there exists a set $\mathcal{I}_2 \subseteq \mathcal{I}_1$ of cardinality at least $\frac{2^{\frac{n}{v^2}}/v^2}{2^{k \log m + k^2 \log 2n + k \cdot r \cdot o(n)}} = 2^{\frac{n}{v^2} - 2 \log v - k \log m - k^2 \log 2n - k \cdot r \cdot o(n)}$, such that for any $\mathbf{D}(I)$ and $\mathbf{D}(J)$ in $\mathcal{I}_2$, we have $\overline{\mathcal{M}^I} = \overline{\mathcal{M}^J}$. For large enough n, we have at least two different instances $\mathbf{D}(I)$ and $\mathbf{D}(J)$ in $\mathcal{I}_2$.

We recall the crucial properties of $\mathbf{D}(I)$ and $\mathbf{D}(J)$:

1. The query RST returns *false* on $\mathbf{D}(I)$ and on $\mathbf{D}(J)$ (cf. $(*)$);
2. M does not check block $B_{i_0}^{j_0}$ on $\mathbf{D}(I)$, nor on $\mathbf{D}(J)$;
3. $\mathbf{D}(I)$ and $\mathbf{D}(J)$ differ on R and T only in block $B_{i_0}^{j_0}$; and
4. For each cursor c, when c has just left block $B_{i_0}^{j_0}$ (on R or T) in the run on $\mathbf{D}(I)$, the machine M is in the same state as when c has just left block $B_{i_0}^{j_0}$ in the run on $\mathbf{D}(J)$.

Let $\mathcal{V}_0, \mathcal{V}_1, \ldots$ be the sequence of states in the run of M on $\mathbf{D}(I)$ and let $\mathcal{W}_0, \mathcal{W}_1, \ldots$ be the sequence of states in the run of M on $\mathbf{D}(J)$. Let t_c^I and t_c^J be the points in time when the cursor c of M has just left block $B_{i_0}^{j_0}$ in the run on $\mathbf{D}(I)$ and $\mathbf{D}(J)$, respectively. Because of Property 4 above, $\mathcal{V}_{t_c^I}$ equals $\mathcal{W}_{t_c^J}$ for each cursor c. Note that the start states $\mathcal{V}_0$ and $\mathcal{W}_0$ are equal.

Now consider instance $\mathbf{D}_{\text{err}} := \mathbf{D}(I, \pi(J)^c)$. So, $\mathbf{D}_{\text{err}}$ has the same lists $R_{\nearrow}, R_{\searrow}$ as $\mathbf{D}(I)$ and the same lists $T_{\nearrow}, T_{\searrow}$ as $\mathbf{D}(J)$. Consider M running on $\mathbf{D}_{\text{err}}$. As long as there are no cursors in block $B_{i_0}^{j_0}$ on R and on T, the machine M running on $\mathbf{D}_{\text{err}}$ will go through the same sequence of states as on $\mathbf{D}(I)$ and as on $\mathbf{D}(J)$. Indeed, M has not yet seen any difference between

$\mathbf{D}_{\text{err}}$ on the one hand, and $\mathbf{D}(I)$ and $\mathbf{D}(J)$ on the other hand (Property 3). At some point, however, there may be some cursor c in block $B_{i_0}^{j_0}$.

- If this is on $R_{\nearrow}$ or $R_{\searrow}$, no cursor on $T_{\nearrow}$ or $T_{\searrow}$ will enter block $B_{i_0}^{j_0}$ as long as c is in this block (Property 2). Therefore, M will go through some successive states $\mathcal{V}_i$ (i.e., M thinks it is working on $\mathbf{D}(I)$) until c has just left block $B_{i_0}^{j_0}$. At that point, M is in state $\mathcal{V}_{t_c^I} = \mathcal{W}_{t_c^J}$ (Property 4) and the machine now again goes through the same sequence of states as on $\mathbf{D}(I)$ and as on $\mathbf{D}(J)$ (Property 3).

- If this is on $T_{\nearrow}$ or $T_{\searrow}$, we are in a similar situation: No cursor on $R_{\nearrow}$ or $R_{\searrow}$ will enter block $B_{i_0}^{j_0}$ as long as c is in this block (Property 2). Therefore, M will go through some successive states $\mathcal{W}_i$ (i.e., M thinks it is working on $\mathbf{D}(J)$) until c has just left block $B_{i_0}^{j_0}$. At that point, M is in state $\mathcal{V}_{t_c^I} = \mathcal{W}_{t_c^J}$ (Property 4) and the machine now again goes through the same sequence of states as on $\mathbf{D}(I)$ and as on $\mathbf{D}(J)$ (Property 3).

Hence, in the run of M on $\mathbf{D}_{\text{err}}$, each time a cursor c has just left block $B_{i_0}^{j_0}$, the machine is in state $\mathcal{V}_{t_c^I}$. Let d be the last cursor that leaves block $B_{i_0}^{j_0}$. When d has just left this block, M is in state $\mathcal{V}_{t_d^I}$. After the last cursor has left block $B_{i_0}^{j_0}$, the run of M on $\mathbf{D}_{\text{err}}$ finishes exactly as the run of M on $\mathbf{D}(I)$ after the last cursor has left block $B_{i_0}^{j_0}$ (and on $\mathbf{D}(J)$ for that matter). In particular, M rejects $\mathbf{D}_{\text{err}}$ because it rejects $\mathbf{D}(I)$ (Property 1). This is wrong, however, because due to $(*)$ the query RST returns *true* on $\mathbf{D}_{\text{err}}$. Finally, this completes the proof of Theorem 5.2. $\square$

Remark 5.3. **(a)** An analysis of the proof of Theorem 5.2 shows that we can make the following, more precise statement: *Let $k, m, r, s : \mathbb{N} \to \mathbb{N}$ such that*

$$k(n)^6 \cdot (\log m(n)) \cdot r(n) \cdot \max(s(n), \log n) = o(n).$$

Then for sufficiently large n, there is no FCM with at most $k(n)$ cursors, $m(n)$ modes, and $r(n)$ registers each holding bitstrings of length at most $s(n)$ that, for all unary relations R, T and binary relations S of size n decides if $R \ltimes_{x_1=y_1} (S \ltimes_{x_2=y_1} T)$ is nonempty. (In the statement of Theorem 5.2, k, m, r are constant.) This is interesting in particular because we can use a substantial number of cursors, polynomially related to the input size, to store data elements and still obtain the lower bound result.

(b) Note that Theorem 5.2 is sharp in terms of arity: if S would have been unary (and R and T of arbitrary arities), then the according RST query would have been computable on sorted inputs.
(c) Furthermore, Theorem 5.2 is also sharp in terms of register bitlength: Assume data elements are natural numbers, and focus on databases with elements from 1 to $O(n)$. If the background provides functions for setting and checking the i-th bit of a bitstring, the query RST is easily computed by an $O(n)$-FCM.

By a variation of the proof of Theorem 5.2 we can also show the following strengthening of Theorem 4.3:

Theorem 5.4. *There is no $o(n)$-FCM working on enumerations of unary relations R and S and their reversals, that checks whether $R \cap S \neq \emptyset$.*

Note that Theorems 5.2 and 5.4 are valid for arbitrary background structures.

6 Concluding Remarks.

A natural question arising from Corollary 4.5 is whether finite cursor machines with sorting are capable of computing relational algebra queries *beyond* the semijoin algebra. The answer is affirmative:

Proposition 6.1. *The boolean query over a binary relation R that asks if $R = \pi_1(R) \times \pi_2(R)$ can be computed by an $O(1)$-FCM working on $\mathsf{sort}_{(1,2),(\nearrow,\nearrow)}(R)$ and $\mathsf{sort}_{(2,1),(\nearrow,\nearrow)}(R)$.*

Proof. The list $\mathsf{sort}_{(1,2),(\nearrow,\nearrow)}(R)$ can be viewed as a list of subsets of $\pi_2(R)$, numbered by the elements of $\pi_1(R)$. The query asks whether all these subsets are in fact equal to $\pi_2(R)$. Using an auxiliary cursor over $\mathsf{sort}_{(2,1),(\nearrow,\nearrow)}(R)$, we check this for the first subset in the list. Then, using two cursors over $\mathsf{sort}_{(1,2),(\nearrow,\nearrow)}(R)$, we check whether the second subset equals the first, the third equals the second, and so on. $\square$

Note that, using an Ehrenfeucht-game argument, one can indeed prove that the query from Proposition 6.1 is not expressible in the semijoin algebra [23].

We have not been able to solve the following:

Open Problem 6.2. *Is there a boolean relational algebra query that cannot be computed by any composition of $O(1)$-FCMs (or even $o(n)$-FCMs) and sorting operations?*

Under a plausible assumption from parameterized complexity theory [10, 8] we can answer the $O(1)$-version of this problem affirmatively for FCMs with a decidable background structure.

There are, however, many queries that are not definable in relational algebra, but computable by FCMs with sorting. By their sequential nature, FCMs can easily compare cardinalities of relations, check whether a directed graph is regular, or do modular counting—and all these tasks are not definable in relational algebra. One might be tempted to conjecture, however, that FCMs with sorting cannot go beyond relational algebra with counting and aggregation, but this is false:

Proposition 6.3. *On a ternary relation G and two unary relations S and T, the boolean query "Check that $G = \pi_{1,2}(G) \times (\pi_1(G) \cup \pi_2(G))$, that $\pi_{1,2}(G)$ is deterministic, and that T is reachable from S by a path in $\pi_{1,2}(G)$ viewed as a directed graph" is not expressible in relational algebra with counting and aggregation, but computable by an $O(1)$-FCM working on sorted inputs.*

Proof. (a): If this query was expressible in relational algebra with counting and aggregation, then deterministic reachability would be expressible, too. However, since deterministic reachability is a non-local query, it is not expressible in first-order with counting and aggregation (see [17]).
(b): A finite cursor machine that solves this query can proceed as follows: The first check follows by Proposition 6.1; the determinism check is easy. The path can now be found using a cursor sorted on the third column of G, which gives us n copies of the graph $\pi_{1,2}(G)$. □

References

[1] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the 44th IEEE Symposium on Foundations of Computer Science*, pages 540–549, 2004.

[2] 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.

[3] M. Altinel and M. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 53–64, 2000.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the 21st ACM Symposium on Principles of Database Systems*, pages 1–16, 2002.

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

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

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

[8] R.G. Downey and M.R. Fellows. *Parameterized Complexity.* Springer, 1999.

[9] R. Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *Journal of the ACM*, 30:514–550, 1983.

[10] J. Flum and M. Grohe. *Parameterized Complexity Theory.* Springer, 2006.

[11] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database System Implementation.* Prentice Hall, 1999.

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

[13] M. Grohe, C. Koch, and N. Schweikardt. Tight lower bounds for query processing on streaming and external memory data. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming*, pages 1076–1088, 2005.

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

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

[16] Y. Gurevich. Evolving algebras 1993: Lipari guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[17] L. Hella, L. Libkin, J. Nurmonen, and L. Wong. Logics with aggregate operators. *Journal of the ACM*, 48(4):880–907, 2001.

[18] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. *External Memory Algorithms. DIMACS Series In Discrete Mathematics And Theoretical Computer Science*, 50:107–118, 1999.

[19] J. Hromkovič. One-way multihead deterministic finite automata. *Acta Informatica*, 19:377–384, 1983.

[20] Y-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 492–503, 2004.

[21] D. Leinders and J. Van den Bussche. On the complexity of division and set joins in the relational algebra. In *Proceedings of the 24th ACM Symposium on Principles of Database Systems*, pages 76–83, 2005.

[22] D. Leinders, M. Marx, J. Tyszkiewicz, and J. Van den Bussche. The semijoin algebra and the guarded fragment. *Journal of Logic, Language and Information*, 14(3):331–343, 2005.

[23] D. Leinders, J. Tyszkiewicz, and J. Van den Bussche. On the expressive power of semijoin queries. *Information Processing Letters*, 91(2):93–98, 2004.

[24] L. Libkin. *Elements of Finite Model Theory.* Springer, 2004.

[25] S. Muthukrishnan. *Data Streams: Algorithms and Applications.* Now Publishers Inc, 2005.

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

[27] A.L. Rosenberg. On multi-head finite automata. In *Proceedings of the 6th IEEE Symposium on Switching Circuit Theory and Logical Design*, pages 221–228, 1965.

[28] D. Simmen, E. Shekita, and T. Malkemus. Fundamental techniques for order optimization. In *Proceedings of the 15th ACM SIGMOD International Conference on Management of Data*, pages 57–67, 1996.

[29] J. Van den Bussche. Finite cursor machines in database query processing. In *Proceedings of the 11th International Workshop on Abstract State Machines*, 2004.

[30] M. Yannakakis. Algorithms for acyclic database schemes. In *Proceedings of the 7th International Conference on Very Large Data Bases*, pages 82–94, 1981.