Theory Comput Syst
DOI 10.1007/s00224-008-9137-7

Database Query Processing Using Finite Cursor Machines

Martin Grohe · Yuri Gurevich · Dirk Leinders · Nicole Schweikardt · Jerzy Tyszkiewicz · Jan Van den Bussche

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.

Keywords Database · Relational algebra · Semijoin · Query processing

1 Introduction

We introduce and analyze *finite cursor machines*, an abstract model of database query processing.[1] 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

[1] The model was first presented in a talk at the ASM 2004 workshop [30].

M. Grohe · N. Schweikardt
Humboldt-University Berlin, Berlin, Germany

Y. Gurevich
Microsoft Research, Redmond, WA, USA

D. Leinders (✉) · J. Van den Bussche
Hasselt University and Transnational University of Limburg, Diepenbeek, Belgium
e-mail: dirk.leinders@uhasselt.be

J. Tyszkiewicz
University of Warsaw, Warsaw, Poland

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. As ASMs are more general than classical automata on strings, they are a better basis to study computational models. The ability to deal with abstract indivisible data elements is important because this is common in database modeling. Furthermore, Gurevich has shown that every sequential algorithm can be modeled as an ASM in a natural way [17].

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, 19, 26]) and the database systems community (e.g., [3, 4, 7, 12, 15, 21, 27]). 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, 23, 31]. 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 query plan composed 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 Sect. 2, the notion of finite cursor machines is introduced in Sect. 3. The power of $O(1)$-FCMs and of $o(n)$-FCMs is investigated in Sects. 4 and 5. Some concluding remarks and open questions can be found in Sect. 6.

2 Preliminaries

Throughout the paper we fix an arbitrary, typically infinite, universe $\mathbb{E}$ of "data elements" equipped with a vocabulary Ω of predicates, including at least the equality predicate. We also fix a database schema $\mathcal{S}$: 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 $\mathbf{D}$. Analogously, a list instance with schema $\mathcal{S}$ assigns to each $R \in \mathcal{S}$ a finite list of k-tuples of data elements, where k is the arity of R.

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 $i_1, \ldots, i_k$. For a relation R with arity n and for a quantifier-free formula $\theta(\overline{x})$ over Ω with variables among $\{x_1, \ldots, x_n\}$, the *selection* operator $\sigma_\theta(R)$ returns those tuples from R that satisfy θ. Finally, for relations R and S with respective arities n and m, and for a quantifier-free formula $\theta(\overline{x}, \overline{y})$ over Ω with variables among $\{x_1, \ldots, x_n, y_1, \ldots, y_m\}$, the *join* operator

$R \bowtie_\theta S$ is defined as $\{(\overline{a}, \overline{b}) : \overline{a} \in R, \overline{b} \in S, \theta(\overline{a}, \overline{b}) \text{ holds}\}$. 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 $\{\overline{a} \in R : \exists \overline{b} \in S : \theta(\overline{a}, \overline{b}) \text{ holds}\}$.

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).[2] 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 detailed 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: Element, 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 Bitstring. In particular, Υ_0 contains all the predicates from Ω (recall beginning of Sect. 2), taken as predicates on the sort Element. 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 $\mathit{attribute}^i_R : \mathsf{Row}_R \to \mathsf{Element}$ for $i = 1, \ldots, k$. Furthermore, we have a constant symbol $\bot_R$ of sort Row_R. Finally, we have a function symbol $\mathit{next}_R : \mathsf{Row}_R \to \mathsf{Row}_R$ 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 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.

[2] 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".

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 is a list instance consisting 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 Element is $\mathbb{E}$; the interpretation of 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 $attribute^i_R$ is defined by $(x_1, \ldots, x_k) \mapsto x_i$, 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 par $r_1 \ldots r_m$ 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 if φ then r_1 else r_2 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 par... 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 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, cannot 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 over $\mathbb{N}$ that returns the sum of the first and second attribute of each row with a third 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 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 5.10).

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 $M_1, \ldots, M_r$ 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 $M_1, \ldots, M_r$, 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 $M_1(\mathbf{D}), \ldots, M_r(\mathbf{D})$.*

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 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 binary relations R and S with condition $x_2 = y_1$ 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 $i = 1, \ldots, w$. 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 $1 \leq i, j \leq w$:

if $mode = check_{i,j}$ and $attribute^2_R(c^i_R) = attribute^1_S(c^j_S)$ then
  par
    $outputmode := out$
    $out_1 := attribute^1_R(c^i_R)$
    $out_2 := attribute^2_R(c^i_R)$
    $out_3 := attribute^1_S(c^j_S)$
    $out_4 := attribute^2_S(c^j_S)$
    $mode := next\text{-}mode_{i,j}$
  endpar
endif

Here, $next\text{-}mode_{i,j}$ is the mode in which the next pair of the w^2 pairs of tuples seen by the cursors is joined. So, if neither i nor j equals w, then $next\text{-}mode_{i,j}$ is either $check_{i,j+1}$ or $check_{i+1,1}$. Next — after *mode* was equal to $check_{w,w}$ — 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.

While we already noted that joins cannot be computed in general by an FCM simply because join outputs can be quadratic in size, we can actually show something much stronger. Indeed, we can show that even checking whether the join is nonempty (so that output size is not an issue) is impossible 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 5.11). 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 [25, Sect. 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 cursor c of M is on a position ℓ on R if M has executed $\ell - 1$ update rules $c := next_R(c)$; a cursor being on a position on S is defined similarly. I.e., if a cursor c is on position ℓ on R (S), then c sees element a_ℓ ($a'_{n-\ell+1}$). We say that a pair of cursors "checks" ℓ if in some state during the run, one of the cursors is on position ℓ on R (i.e., the cursor sees element a_ℓ) and the other cursor is on position $n - \ell + 1$ on S (i.e., the cursor sees element a'_ℓ). By the way the lists are ordered, every pair of cursors can check only one ℓ. Hence, some i is not

checked. Now replace a'_i in S by a_i, obtaining set S', and consider the run of M on R and S'. Because the element a'_i has the same relative order as a_i with respect to the other elements in the lists, any tuple of elements will satisfy the same predicates as the tuple obtained by replacing a_i by a'_i. The run of M on R and S' will thus be the same as the run of M on R and S. The intersection of R and S, however, is empty, while the intersection of R and S' is not. So, M cannot exist. □

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. The same holds for the difference $R - S$. Moreover, while the full join is still not computable by an FCM working on sorted inputs, simply because the output size can be too large, *semijoins* $R \ltimes_\theta S$ now become also computable by FCMs on sorted inputs. Specifically, this will be possible for a class of "allowed" join conditions θ which we define next.

Definition 4.4 Recall the definition of a join condition $\theta(x_1, \ldots, x_n, y_1, \ldots, y_m)$ from Sect. 2, and assume that the vocabulary Ω includes a total order $<$ on $\mathbb{E}$. We say that θ is *allowed* if it is of the form $\varphi \wedge \psi$, where φ is a conjunction of equalities, and where ψ is a conjunction of at most two inequalities of the form $x_i < y_j$ or $x_i > y_j$. (As special cases, φ and/or ψ can simply be true.)

When ψ is not of the form $x_i < y_j \wedge x_k < y_l$ (i.e., two "less than" predicates between an x and a y), we call θ *A-allowed*; otherwise θ is called *AD-allowed*. In this, the "A" stands for ascending and the "D" stands for descending.

In the following examples we will show how A-allowed semijoins can be computed by $O(1)$-FCMs on sorted inputs. The AD-allowed case will be discussed in the following section.

Example 4.5 Let R and S be binary relations and consider the semijoin $R \ltimes_\theta S$, where θ is the A-allowed join condition $x_1 = y_1 \wedge x_2 > y_2$. The FCM computing this semijoin works by doing a synchronized scan of R and S sorted on their respective first columns. Suppose for a tuple $\overline{r}$ in R, a tuple $\overline{s}$ in S is found with $r_1 = s_1$, i.e., the first component of $\overline{r}$ equals the first component of $\overline{s}$. Then, the FCM searches for the minimum value for s'_2 of all tuples $\overline{s}'$ in S with $s'_1 = s_1 (= r_1)$; note that these tuples occur in a contiguous region following $\overline{s}$ in S. We denote this minimum value by v. Then, a cursor on R visits all tuples $\overline{r}'$ with $r'_1 = r_1$ and outputs all of these tuples having $r'_2 > v$. Again, note that these tuples occur in a contiguous region following $\overline{r}$ in R. Then, the next tuple $\overline{r}$ in R is considered. And so on.

When θ is the condition $x_1 = y_1 \wedge x_2 < y_2$, the semijoin is computed using a similar strategy, except that instead of the minimum value, here the maximum value for s'_2 is searched and the tuples $\overline{r}'$ in R with $r'_2 < v$ are output.

Example 4.6 Let R and S be ternary relations and consider the semijoin $R \ltimes_\theta S$, where θ is the A-allowed join condition $x_1 = y_1 \wedge x_2 > y_2 \wedge x_3 > y_3$. The FCM computing this semijoin works on R and S sorted lexicographically on their respective first columns first, and on their second columns second. Again, the FCM first

searches for a tuple $\overline{r}$ in R for which there exists a tuple $\overline{s}$ in S with $r_1 = s_1$. Then, the FCM searches the first tuple $\overline{s}'$ in S with $s'_1 > s_1 (= r_1)$ or $s'_2 \geq r_2$ and searches for the minimum value for s''_3 of all tuples $\overline{s}''$ in the region starting at $\overline{s}$ and ending at (not including) $\overline{s}'$. We denote this minimum value by v. Then, a cursor on R visits all tuples $\overline{r}'$ with $r'_1 = r_1$ and $r'_2 = r_2$. Of these tuples the ones with $r'_3 > v$ are output.

While visiting the tuples $\overline{r}'$, three things can occur: (1) the end of R is reached; (2) a tuple $\overline{r}''$ is found with $r''_2 > r_2$; or (3) a tuple $\overline{r}''$ is found with $r''_1 > r_1$. In case (1), the FCM stops. In case (2), the cursor that was positioned at $\overline{s}'$ is moved forward to search again for the first tuple $\overline{s}'$ with $s'_1 > s_1 (= r_1)$ or $s'_2 \geq r''_2$. Also, the minimum value v for s''_3 of all tuples $\overline{s}''$ in the region between $\overline{s}$ and the new $\overline{s}'$ is updated. Note that this region grows. Again, a cursor on R visits all tuples $\overline{r}'$ with $r'_1 = r_1$ and $r'_2 = r''_2$ and the ones with r'_3 strictly greater than v are output. Finally, in case (3), the FCM starts searching again for a tuple $\overline{s}$ with $s_1 = r''_1$. And so on.

When θ is the condition $x_1 = y_1 \wedge x_2 > y_2 \wedge x_3 < y_3$, the semijoin is computed using a similar strategy, except that instead of the minimum value, here the maximum value for s''_3 is searched and the tuples $\overline{r}'$ in R with $r'_3 < v$ are output.

Note that also semijoins where the condition θ is a disjunction of allowed join conditions can be computed by an FCM on sorted inputs by computing the semijoins with condition φ for each allowed join condition φ in the disjunction θ and then computing the union of these results.

The easy observations above 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, when only considering sorting in ascending order, 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's $\rho(1)$-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. In this section, we only consider sorting in ascending order. In the next section we also consider sorting in descending order: we agree that by an FCM working on sorted inputs, we always mean that inputs are sorted in ascending order, unless we state otherwise. We then summarize the above discussion as follows:

Proposition 4.7 *Each operator of the semijoin algebra* (*i.e*, *union*, *intersection*, *difference*, *projection*, *selection*, *and semijoin with A-allowed join condition*) *can be computed by an* $O(1)$*-FCM on sorted inputs.*

Proof We only consider the semijoin operator with A-allowed join condition. The other operators can easily be computed by an $O(1)$-FCM on sorted inputs.

Let $R \ltimes_\theta S$ be a semijoin with A-allowed join condition θ. Recall that θ is of the form $\varphi \wedge \psi$. We consider three cases depending on whether ψ consists of zero, one, or two inequalities: (1) ψ is true, (2) ψ is either $x_{\psi^1} < y_{\psi^2}$ or $x_{\psi^1} > y_{\psi^2}$ for some ψ^1 and ψ^2, and (3) ψ is either $x_{\psi_1^1} > y_{\psi_1^2} \wedge x_{\psi_2^1} < y_{\psi_2^2}$ or $x_{\psi_1^1} > y_{\psi_1^2} \wedge x_{\psi_2^1} > y_{\psi_2^2}$ for some ψ_1^1, ψ_1^2, ψ_2^1 and ψ_2^2. Let φ be $\bigwedge_{\ell=1}^{k} x_{\varphi_\ell^1} = y_{\varphi_\ell^2}$. Without loss of generality, we assume that columns participating in an inequality do not participate in an equality.

I.e., in case (2), $\psi^i \notin \{\varphi^i_\ell \mid \ell = 1, \ldots, k\}$ for $i = 1, 2$, and in case (3) $\psi^i_1, \psi^i_2 \notin \{\varphi^i_\ell \mid \ell = 1, \ldots, k\}$ for $i = 1, 2$.

We describe how an FCM M computes $R \ltimes_\theta S$. In case (1), M works on relation R (S) sorted on its φ^1_1-th (φ^2_1-th) column first, φ^1_2-th (φ^2_2-th) column second, and φ^1_k-th (φ^2_k-th) column k-th. Then, M performs a synchronized scan to search a joining tuple in S for each tuple in R.

In case (2), M works on R and S sorted as before. We first consider the subcase where ψ is $x_{\psi^1} < y_{\psi^2}$. The machine considers each tuple $\overline{r}$ in R in turn and searches for the set of tuples $\overline{s}$ in S such that $\varphi(\overline{r}, \overline{s})$ is true. Note that these tuples occur in a contiguous region in S and that the machine can mark this region using two cursors, one pointing to the first such tuple and one pointing to the tuple right below the last such tuple. The region is illustrated in Fig. 1. Then, M searches for the maximum value for column ψ^2 in this region. In Fig. 1, this value is denoted $w_{\max}$. The machine now outputs all tuples in relation R with the same elements as $\overline{r}$ in columns $\varphi^1_1, \ldots, \varphi^1_k$ and with an element u in column ψ^1 less than $w_{\max}$. In the subcase where ψ is $x_{\psi^1} > y_{\psi^2}$, the machine searches for the *minimum* value $w_{\min}$ for column ψ^2 in the above region and outputs all tuples in relation R with the same elements as $\overline{r}$ in columns $\varphi^1_1, \ldots, \varphi^1_k$ and with an element u in column ψ^1 *greater* than $w_{\min}$.

In case (3), M works on relation R (S) sorted on its φ^1_1-th (φ^2_1-th) column first, φ^1_2-th (φ^2_2-th) column second, φ^1_k-th (φ^2_k-th) column k-th, and finally on its ψ^1_1-th (ψ^2_1-th) column. We first consider the subcase where ψ is $x_{\psi^1_1} > y_{\psi^2_1} \wedge x_{\psi^1_2} < y_{\psi^2_2}$. The machine considers each tuple $\overline{r}$ in R in turn and searches for the set of tuples $\overline{s}$ in S such that $\varphi(\overline{r}, \overline{s})$ is true and additionally, the ψ^1_1-th element of $\overline{r}$ is greater than the ψ^2_1-th element of $\overline{s}$. Note that these tuples occur in a contiguous region in S, which we will denote region($\overline{r}$), and that the machine can mark this region using two cursors, one pointing to the first such tuple and one pointing to the tuple right below

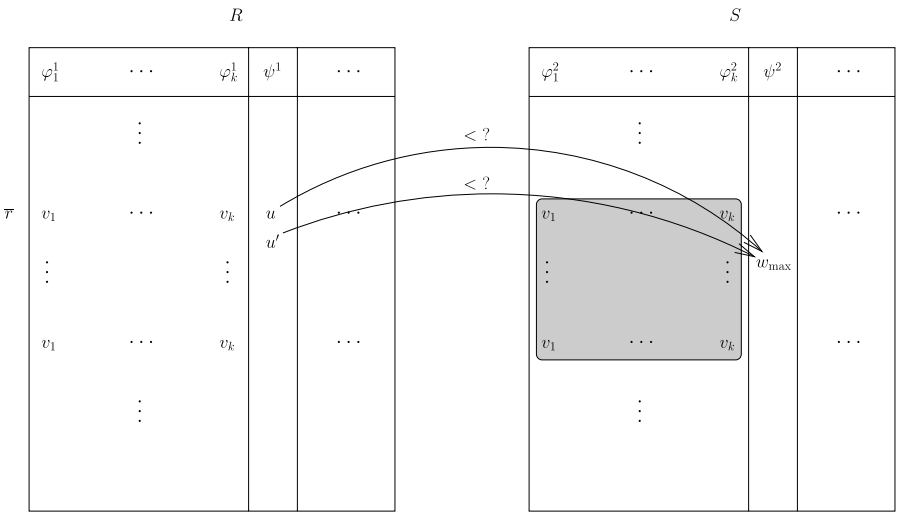

Fig. 1 Illustrating the algorithm in the proof of Proposition 4.7 (case (2))

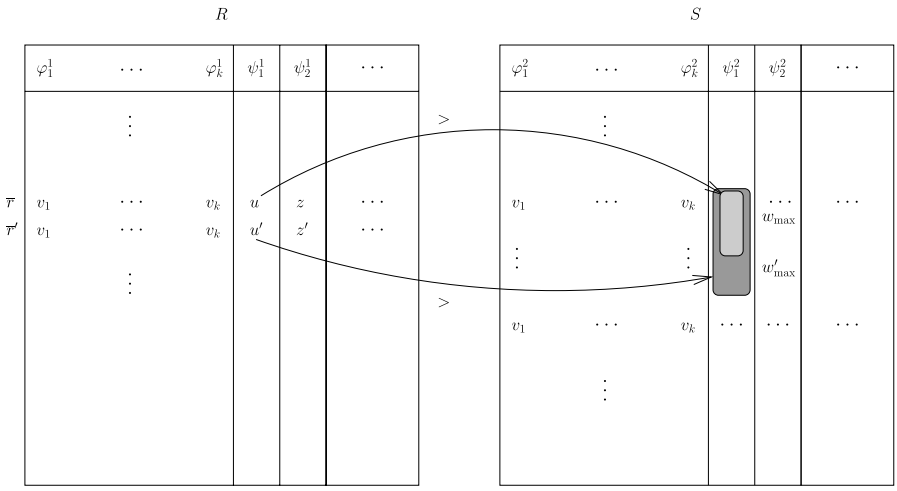

Fig. 2 Illustrating the algorithm in the proof of Proposition 4.7 (case (3))

the last such tuple. The region region($\overline{r}$) is shown as a light gray rectangle in Fig. 2. Then, M searches for the maximum value for column ψ_2^2 in this region. In Fig. 2, this value is denoted $w_{\max}$. The machine now outputs all tuples in relation R with the same elements as $\overline{r}$ in columns $\varphi_1^1, \ldots, \varphi_k^1, \psi_1^1$ and with an element z in column ψ_2^1 less than $w_{\max}$. Note that the region region($\overline{r}'$) of a tuple $\overline{r}'$ following $\overline{r}$ and with the same elements as $\overline{r}$ in columns $\varphi_1^1, \ldots, \varphi_k^1$ will contain region($\overline{r}$). The region region($\overline{r}'$) is shown as a dark gray rectangle in Fig. 2. The maximum value $w'_{\max}$ for column ψ_2^2 in this region will therefore be a least as great as $w_{\max}$. Hence, the cursor pointing to that maximum value can be easily updated. The machine now outputs all tuples in relation R with the same elements as $\overline{r}'$ in columns $\varphi_1^1, \ldots, \varphi_k^1, \psi_1^1$ and with an element z' in column ψ_2^1 less than $w'_{\max}$. The machine continues in this way. The subcase where ψ is $x_{\psi_1^1} > y_{\psi_1^2} \wedge x_{\psi_2^1} > y_{\psi_2^2}$ is handled similarly. □

Corollary 4.8 *Every semijoin algebra query with A-allowed join conditions can be computed by a query plan composed of $O(1)$-FCMs and sorting operations.*

Proof From the expression tree of the given semijoin algebra expression we construct a query plan as follows: we replace each selection, projection and union operator by an FCM computing that operator; we replace each intersection, difference and semijoin operator by an FCM computing that operator *on sorted inputs*; and finally, we insert sorting operations so that the FCMs computing intersection, difference and semijoin have access to all possible sorted orders. □

The following example illustrates the construction in the proof of Corollary 4.8.

Example 4.9 Consider the query $Q := (\sigma_{x_1 < y_2}(R \cup S)) \ltimes_{x_2 = y_2} \sigma_{x_2 < y_1} T$. The expression tree of Q is shown in Fig. 3 on the left. The query plan obtained by using the

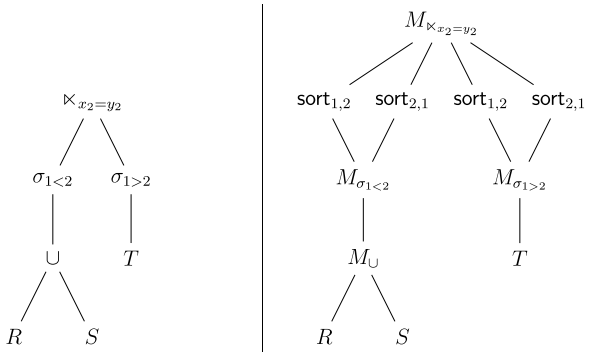

Fig. 3 *On the left*: expression tree for query $Q = (\sigma_{x_1<y_2}(R \cup S)) \ltimes_{x_2=y_2} \sigma_{x_1>y_2} T$. *On the right*: query plan computing Q, composed of $O(1)$-FCMs and sorting operations

construction in the proof of Corollary 4.8 is shown on the right of Fig. 3. Here, M_α is used to denote the $O(1)$-FCM computing the operator α.

The simple proof of Corollary 4.8 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 [29].

Example 4.10 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 naive 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 question 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

$$e_1 := E - (E \underset{\substack{x_2=y_1 \\ x_1=y_2}}{\ltimes} E),$$

$$e_2 := E \underset{\substack{x_2=y_1 \\ x_1 \neq y_2}}{\ltimes} E,$$

$$e_3 := (E \underset{x_1=y_1}{\ltimes} C) \underset{x_2=y_1}{\ltimes} ((\pi_1(E) \cup \pi_2(E)) - C).$$

Here, expression e_1 selects the edges that do not have a reverse edge; expression e_2 selects the edges that have a follow-up edge; and expression e_3 selects the edges whose end points have different colors. The semijoin $E \ltimes_{\substack{x_2=y_1 \\ 1 \neq y_2}} E$ is an abbreviation for the union of the allowed semijoins $E \ltimes_{\substack{x_2=y_1 \\ x_1<y_2}} E$ and $E \ltimes_{\substack{x_2=y_1 \\ x_1>y_2}} E$.

Before proving that the monochromatic 2-cycle query cannot 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 [28].

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

$$L_n^f := \big\{ w_1 b w_2 b \dots b w_f b w'_f b \dots b w'_2 b w'_1 \mid \forall i = 1, \dots, f : w_i, w'_i \in \{a,b\}^* \text{ and } |w_i| = |w'_i| = n \big\}$$

$$P_n^f := \big\{ w_1 b w_2 b \dots b w_f b w'_f b \dots b w'_2 b w'_1 \in L_n^f \mid \forall i = 1, \dots, f : w_i^R = w'_i \big\}.$$

We recall the following result:

Lemma 4.11 (Hromkovič [20]) *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 $L_n^f - P_n^f$.*

Actually, we will need a slight strengthening of the above Lemma, which can be proven in exactly the same way as Lemma 4.11. To make this paper self-contained and also for easy reference, we still provide a polished proof below. The strengthening deals with *oblivious right-to-left heads* that can only move from right to left on the input tape sensing other heads, but cannot read the symbols on the tape.

Lemma 4.12 *Let M be a one-way, k-head, sensing DFA with oblivious right-to-left heads, 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 sequence of "prominent" configurations of M, where a prominent configuration is a halting one, or one in which a left-to-right 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. Any given run of M has at most $2fk$ prominent configurations, so there are at most

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

different sequences of prominent configurations. As there are 2^{fn} different strings in P_n^f, there is a set G of at least $2^{fn}/p(n)$ different strings in P_n^f with the same sequence of prominent configurations.

On any $w_1 b w_2 b \ldots b w_f b w_f^R b \ldots b w_2^R b w_1^R \in P_n^f$, we say that M "checks" region $i \in \{1, \ldots, f\}$ if at some point during the run, there is a left-to-right head in w_i, and another left-to-right head in w_i^R. Every pair of left-to-right heads can check at most one i, so since $f > \binom{k}{2}$, at least one i is not checked.

In our set G, the non-checked i is the same for all strings, because they have the same sequence of prominent configurations. 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 subset 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

$$y_1 = w_1 b w_2 b .. b w_i b .. b w_n b w_n^R b .. b w_i^R b .. b w_2^R b w_1^R,$$
$$y_2 = w_1 b w_2 b .. b w_i' b .. b w_n b w_n^R b .. b w_i'^R b .. b w_2^R b w_1^R$$

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

$$w_1 b w_2 b \ldots b w_i b \ldots b w_n b w_n^R b \ldots b w_i'^R b \ldots b w_2^R b w_1^R.$$

Indeed, while a left-to-right head of M is in w_i, no left-to-right head is in w_i^R and thus the run behaves as on y_1; while a left-to-right head of M is in $w_i'^R$, no left-to-right head is in w_i and thus the run behaves as on y_2. Since y_1 and y_2 have the same sequence of prominent configurations, y has that sequence as well and hence y is accepted. □

We are now able to prove:

Theorem 4.13 *The monochromatic 2-cycle query is not computable by an $O(1)$-FCM on sorted inputs.*

Proof Note that as a corollary of Lemma 4.12, 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 the predicate $<$ in the conditions of its if-then-else 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. Also note that for this particular relation E, a cursor on $\mathsf{sort}_{2,1}E$ can be simulated by a cursor on $\mathsf{sort}_{1,2}E$ by simply switching the roles of the first and second component. We can thus assume that M' has no cursors on $\mathsf{sort}_{2,1}E$. From FCM M' we can construct a 1DSeFA(k) with oblivious right-to-left heads that would recognize P as follows:

- each cursor on $\mathsf{sort}_{1,2}E$ corresponds to a pair consisting of a "normal" left-to-right head and an oblivious right-to-left head;
- each cursor on $\mathsf{sort}_1 C$ corresponds to a normal head;
- each time a cursor on $\mathsf{sort}_{1,2}E$ is advanced, the normal head of the corresponding pair of heads is moved one position to the right and the oblivious head is moved one position to the left;
- each time a cursor on $\mathsf{sort}_1 C$ is advanced, the corresponding head is moved to the next 1 on the input tape;
- the finite state of the automaton keeps track of the mode of the finite cursor machine, together with the relative positions of all heads. Note that for example the element in the second component of a tuple in $\mathsf{sort}_{1,2}E$ seen by cursor c is lower than the element seen by cursor c' on $\mathsf{sort}_1 C$ if and only if the oblivious right-to-left head corresponding to c is on a position in w before the normal head corresponding to c';
- conditions in if-then-else rules of M' are evaluated by examining the finite state of the automaton.

We conclude that FCM M cannot exist. □

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 4.13 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 $f: \{1, \ldots, p\} \to \{\nearrow, \searrow\}$ indicates ascending or descending. To distin-

guish 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 4.13. We have seen a special case of the monochromatic 2-cycle query there and we showed that it cannot be computed by an $O(1)$-FCM working on ascendingly sorted inputs. It is easy to see that special case can be computed by an $O(1)$-FCM working on both ascendingly and descendingly sorted inputs. Specifically, the "Palindrome" query about a binary relation R and a unary relation C asks whether R is of the form $\{(a_i, a_{n-i+1}) \mid i = 1, \ldots, n\}$ with $a_1 < \cdots < a_n$, and $C \subseteq \{a_1, \ldots, a_n\}$ 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). We thus have:

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.*

Using descending sorting, we can also compute semijoins with AD-allowed join conditions (recall Definition 4.4):

Proposition 5.2 *Every semijoin operation with AD-allowed join condition can be computed by an $O(1)$-FCM on AD-sorted inputs.*

Proof We only consider the semijoin operator with AD-allowed join condition. We have already showed in Proposition 4.7 that the other operators and the semijoin operator with A-allowed join condition can be computed by an $O(1)$-FCM on sorted inputs.

Let $R \ltimes_\theta S$ be a semijoin with AD-allowed join condition θ. Recall that θ is of the form $\varphi \wedge \psi$ where ψ is $x_{\psi_1^1} < y_{\psi_1^2} \wedge x_{\psi_2^1} < y_{\psi_2^2}$ for some ψ_1^1, ψ_1^2, ψ_2^1 and ψ_2^2. As in case (3) of the proof of Proposition 4.7, M works on relation R (S) sorted on its φ_1^1-th (φ_1^2-th) column first, φ_2^1-th (φ_2^2-th) column second, φ_k^1-th (φ_k^2-th) column k-th (all ascending), and finally on its ψ_1^1-th (ψ_1^2-th) column sorted *desendingly*. Now, M computes similarly as in case (3) of the proof of Proposition 4.7. □

We illustrate the algorithm in the proof Proposition 5.2 with an example.

Example 5.3 Let R and S be binary relations and consider the semijoin $R \ltimes_\theta S$, where θ is $x_1 < y_1 \wedge x_2 < y_2$. The FCM computing this semijoin works on R and S sorted descendingly on their respective first columns. For each tuple $\overline{r}$ in R, the set of tuples $\overline{s}$ in S with $s_1 > r_1$ is a contiguous region starting from the first tuple in S until right before the first tuple $\overline{s}'$ with $s_1' \leq r_1$. The FCM then searches for the maximum value for s_2'' of all tuples $\overline{s}''$ in this region. We denote this maximum by v. A cursor on R visits all tuples $\overline{r}'$ with $r_1' = r_1$ and outputs the ones with $r_2' < v$.

While visiting the tuple $\overline{r}'$, two things can occur: (1) the end of R is reached, or (2) a tuple $\overline{r}''$ is found with $r_1'' < r_1$. In case (1), the FCM stops. In case (2), the cursor

that was positioned at $\overline{s}'$ is moved forward to search again for the first tuple $\overline{s}'$ with $s'_1 \leq r''_1$. Also, the maximum value v for s''_2 of all tuples $\overline{s}''$ in the region between the first tuple in S and the new $\overline{s}'$ is updated. Note that this region grows. The machine continues in this way.

As a corollary, we have (cf. Corollary 4.8):

Corollary 5.4 *Every semijoin algebra query with AD-allowed join conditions can be computed by a query plan composed of $O(1)$-FCMs and ascending and descending sorting operations.*

Remark 5.5 We should note that our notion of allowed join condition (Definition 4.4) probably does not exhaust all possible kinds of semijoins that can be computed on AD-sorted inputs. It is indeed conceivable that certain predicates other than equalities and inequalities might exist for which the sorting order of the inputs can still be exploited for computing the semijoin by an FCM.

Moreover, it remains open to prove that semijoins with non-allowed join conditions that involve only $<$ are *not* computable by an FCM on AD-sorted inputs. For example, we conjecture that $R \ltimes_{\substack{x_1<y_1 \\ x_2<y_2 \\ x_3<y_3}} S$, for ternary relations R and S, is not computable by an FCM on AD-sorted inputs.

5.1 Intermediate Sorting Cannot Be Avoided

We now return to the issue of intermediate sorting and establish our main result.

The result will follow from two lemmas, which we state and prove first. In both lemmas, FCMs will work on lists of tuples and their reversals. With respect to sorting, the connection between a list and its reversal is clear: the reversal of an ascendingly sorted list L is the list L sorted descendingly, and vice versa. The first lemma concerns the inherent limitations of FCMs due to the one-way nature of the cursors. In order to state it, we need to define a number of notions. First, for natural numbers v and n with n a multiple of v^2, 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 $\{(i-1)\frac{n}{v} + 1, \ldots, i\frac{n}{v}\}$. Then, further subdivide each block B_i evenly in v consecutive subblocks, denoted by $B_i^1, \ldots, B_i^v$. So, B_i^j equals the set $\{(i-1)\frac{n}{v} + (j-1)\frac{n}{v^2} + 1, \ldots, (i-1)\frac{n}{v} + j\frac{n}{v^2}\}$.

Furthermore, consider the following permutation of $\{1, \ldots, n\}$:

$$\pi_{n,v}\colon\ (i-1)\cdot\frac{n}{v} + s \mapsto (v-i)\cdot\frac{n}{v} + s$$

for $1 \leq i \leq v$ and $1 \leq s \leq \frac{n}{v}$. So, $\pi_{n,v}$ maps subset B_i to subset B_{v-i+1}, and $\pi_{n,v}$ maps subset B_i^j to subset B_{v-i+1}^j. The permutation $\pi_{n,v}$ reverses the blocks B_i in order but it does not reverse the blocks B_i^j inside B_i in order. The permutation $\pi_{16,4}$ is shown in Fig. 4.

Let M be a FCM with k cursors. Let $v = \binom{k}{2} + 1$ and let n be a multiple of v^2. Suppose M works on a set of lists and their reversals. The reversal of a list L is

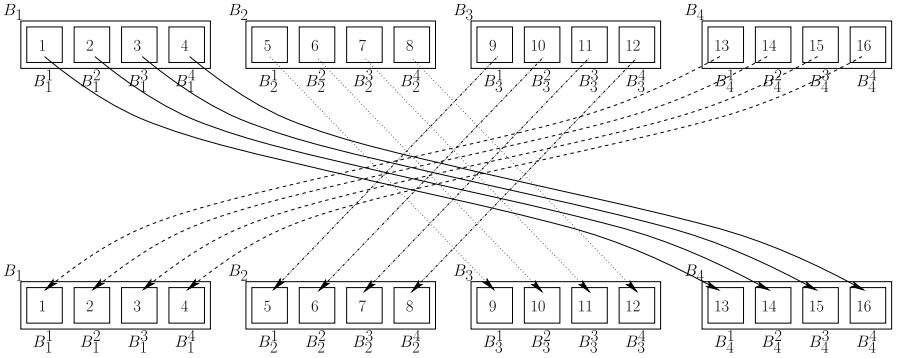

Fig. 4 Permutation $\pi_{16,4}$

denoted by $\overleftarrow{L}$. Consider two distinguished lists L_1 and L_2 of length n on which M is working. In particular, for a clear understanding, let L_1 be the list $t_1^1 \dots t_n^1$ and let L_2 be the list $t_1^2 \dots t_n^2$ for some tuples $t_1^1 \dots t_n^1, t_1^2 \dots t_n^2$.

Consider the run of M on the lists and their reversals. We say that a cursor c is on position ℓ on list L if it has executed $\ell - 1$ update rules $c := next_L(c)$. I.e., if cursor c is on position ℓ on L_1, then c sees tuple t_ℓ^1. We use analogous notation for the lists $\overleftarrow{L_1}$, L_2, and $\overleftarrow{L_2}$. I.e., if a cursor c is on position ℓ on $\overleftarrow{L_1}$ (resp. L_2, resp. $\overleftarrow{L_2}$), then c sees tuple $t_{n-\ell+1}^1$ (resp. t_ℓ^2, resp. $t_{n-\ell+1}^2$). We say that a pair of cursors of M *checks block* B_i if at some state during the run either:

- one cursor in the pair is on a position in B_i on L_1 (i.e., the cursor sees a tuple t_ℓ^1, for some $\ell \in B_i$) and the other cursor in the pair is on a position in B_{v-i+1} on L_2 (i.e., the cursor sees a tuple $t_{\pi\ell}^2$, for some $\ell \in B_i$), or
- one cursor in the pair is on a position in B_{v-i+1} on $\overleftarrow{L_1}$ (i.e., the cursor sees a tuple t_ℓ^1, for some $\ell \in B_i$) and the other cursor in the pair is on a position in B_i on $\overleftarrow{L_2}$ (i.e., the cursor sees a tuple $t_{\pi\ell}^2$, for some $\ell \in B_i$).

Note that each pair of cursors working on the lists L_1 and L_2 or on the lists $\overleftarrow{L_1}$ and $\overleftarrow{L_2}$, 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 L_1 and L_2 or on $\overleftarrow{L_1}$ and $\overleftarrow{L_2}$. We now define the notion of a pair of cursors checking a subblock B_i^j, analogously to the notion of a pair of cursors checking a block B_i. We say that a pair of cursors of M *checks subblock* B_i^j if at some state during the run either:

- one cursor in the pair is on a position in B_i^j on L_1 (i.e., the cursor sees a tuple t_ℓ^1, 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 $\overleftarrow{L_2}$ (i.e., the cursor sees a tuple $t_{\pi\ell}^2$, 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 $\overleftarrow{L_1}$ (i.e., the cursor sees a tuple t_ℓ^1, 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 L_2 (i.e., the cursor sees a tuple $t_{\pi\ell}^2$, for some $\ell \in B_i^j$).

Note that each pair of cursors working either on L_1 and $\overleftarrow{L_2}$ or on $\overleftarrow{L_1}$ and L_2, 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 L_1 and $\overleftarrow{L_2}$ or on $\overleftarrow{L_1}$ and L_2. Note that, since the entire block B_{i_0} is not checked by any pair or cursors working either on L_1 and L_2 or on $\overleftarrow{L_1}$ and $\overleftarrow{L_2}$, the subblock $B_{i_0}^{j_0}$ is thus not checked by *any* pair of cursors (on L_1, $\overleftarrow{L_1}$, L_2, $\overleftarrow{L_2}$).

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

The above argument thus leads to the following:

Lemma 5.6 (Block-checking lemma) *Let M be an FCM with k cursors working on a set of lists and their reversals. Let $v = \binom{k}{2} + 1$ and let n be a multiple of v^2. Let L_1 and L_2 be two distinguished length-n lists in terms of which the notion of "checking a (sub)block" is defined.*

Then, there is at least one subblock $B_{i_0}^{j_0}$ that M does not check.

The block-checking lemma is a building block in the proof of the next lemma, from which our main result will be proved. In order to state the lemma, we need a definition.

Definition 5.7 (Binary (n, v)-collection with respect to (L_1, L_2)) Let n and v be natural numbers such that n is a multiple of v^2. Let $\mathcal{S}$ be a database schema and let L_1 and L_2 be two distinguished relation names in $\mathcal{S}$. A collection $\mathcal{L}$ of list instances with schema $\mathcal{S}$ is called a *binary (n, v)-collection with respect to* (L_1, L_2) if $\mathcal{L}$ is of the form $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$ for which there exist elements $x_1, \ldots, x_n$, $x'_1, \ldots, x'_n$, $y_1, \ldots, y_n$, and $y'_1, \ldots, y'_n$ with $x_i \neq x'_i$ and $y_i \neq y'_i$ for $i = 1, \ldots, n$ such that:

- the lists L_1 and L_2 in list instance $\mathbf{L}_n(I)$ have length n; and
- the i-th element of list L_1 in list instance $\mathbf{L}_n(I)$ is

$$\begin{cases} x_i & \text{if } i \in I, \\ x'_i & \text{if } i \in I^c, \end{cases}$$

  where the complement I^c is taken with respect to $\{1, \ldots, n\}$; and
- the $\pi_{n,v}(i)$-th element of list L_2 in list instance $\mathbf{L}_n(I)$ is

$$\begin{cases} y'_i & \text{if } i \in I, \\ y_i & \text{if } i \in I^c, \end{cases}$$

- the lists other than L_1 and L_2 in $\mathbf{L}_n(I)$ do not depend on I. In particular, they are the same in every instance $\mathbf{L}_n(I)$ of $\mathcal{L}$. And finally,
- the length of the lists other than L_1 and L_2 in $\mathbf{L}_n(I)$ is bounded by αn for some fixed α, independent of n.

We call a binary (n, v)-collection with respect to (L_1, L_2) "binary" because for any $i = 1, \ldots, n$ and for any list instance $\mathbf{L}_n(I)$ in the collection, on the i-th position of L_1 (and L_2) in $\mathbf{L}_n(I)$, there can only be two elements.

Lemma 5.8 (Fooling lemma) *Let M be a $o(n)$-FCM with k cursors. Let $v = \binom{k}{2} + 1$ and let n be a sufficiently large multiple of v^2. If $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$ is a binary (n, v)-collection with respect to (L_1, L_2), then there exist $I, J \subseteq \{1, \ldots, n\}$ with $I \neq J$ such that the run of M working on the list instance containing*:

- *the list L_1 of $\mathbf{L}_n(I)$,*
- *the list L_2 of $\mathbf{L}_n(J)$,*
- *the lists other than L_1 and L_2 of $\mathbf{L}_n(I)$, and*
- *all reversals of the aforementioned lists*

ends in exactly the same way as the run of M on the lists in $\mathbf{L}_n(I)$ and their reversals.

Proof 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 r be the number of registers and let m be the number of modes occurring in M's program.

The proof now consists of two arguments: a counting argument and a fooling argument. The counting argument gives us two instances $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ with $I \neq J$ such that the runs of M on the lists in both instances and their reversals are very "similar". In the fooling argument, it is shown that the instances $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ can be combined into an instance $\mathbf{L}_{\text{err}}$ (in the way defined in the statement of this lemma) such that the run of M on the lists in $\mathbf{L}_{\text{err}}$ and their reversals ends in exactly the same way as the run of M on the lists in $\mathbf{L}_n(I)$ and their reversals.

In the rest of this proof, when considering the run of M on an instance $\mathbf{L}$, we implicitly mean the run of M on the lists in $\mathbf{L}$ *and their reversals.*

A. *Counting argument* Consider the set $\mathcal{I}$ of 2^n instances $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$. According to the block-checking Lemma 5.6, where block-checking is defined in terms of the lists L_1 and L_2, on each instance $\mathbf{L}_n(I)$ 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 $\mathcal{I}_0$.

At this point it is useful to introduce the following terminology. By "block $B_{i_0}^{j_0}$ on L_1", we refer to the positions in $B_{i_0}^{j_0}$ of list L_1 and to the positions in $B_{v-i_0+1}^{v-j_0+1}$ of list $\overleftarrow{L_1}$, i.e., "block $B_{i_0}^{j_0}$ on L_1" contains elements x_ℓ or x'_ℓ where $\ell \in B_{i_0}^{j_0}$. By "block $B_{i_0}^{j_0}$ on L_2", however, we refer to the positions in $B_{v-i_0+1}^{j_0}$ of list L_2 and to the positions in $B_{i_0}^{v-j_0+1}$ of list $\overleftarrow{L_2}$, i.e., "block $B_{i_0}^{j_0}$ on L_2" contains elements $y_{\pi_{n,v}\ell}$ or $y'_{\pi_{n,v}\ell}$ 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".

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

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

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 $\frac{2^n/v^2}{2^{n-n/v^2}} = 2^{n/v^2}/v^2$, such that for any $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ 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, $2^{n/v^2}/v^2$ becomes arbitrarily large.

Let $\mathbf{L}_n(I)$ be an element of $\mathcal{I}_1$. Consider the run of M on $\mathbf{L}_n(I)$. Let c be a cursor and let $\mathcal{M}_c^I$ be the state of M in the run on $\mathbf{L}_n(I)$ when cursor c has just left block $B_{i_0}^{j_0}$ on L_1 or on L_2. Let $\overline{\mathcal{M}^I}$ be the k-tuple consisting of these states $\mathcal{M}_c^I$ for all cursors c. 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 αn possible positions), and the contents of the r bitstring registers (each of which has length $o(n)$). Hence, there are only $m \cdot (\alpha n)^k \cdot 2^{r \cdot o(n)}$ different states for M. The tuple $\overline{\mathcal{M}^I}$ can thus have only

$$(m \cdot (\alpha n)^k \cdot 2^{r \cdot o(n)})^k = 2^{k \log m + k^2 \log \alpha n + k \cdot r \cdot o(n)}$$

different values.

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 $\frac{2^{n/v^2}/v^2}{2^{k \log m + k^2 \log \alpha n + k \cdot r \cdot o(n)}} = 2^{\frac{n}{v^2} - 2\log v - k \log m - k^2 \log \alpha n - k \cdot r \cdot o(n)}$, such that for any $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ 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 $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ in $\mathcal{I}_2$.

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

1. M does not check block $B_{i_0}^{j_0}$ on $\mathbf{L}_n(I)$, nor on $\mathbf{L}_n(J)$;
2. $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ differ on L_1 and L_2 only in block $B_{i_0}^{j_0}$; and
3. For each cursor c, when c has just left block $B_{i_0}^{j_0}$ (on L_1 or L_2) in the run on $\mathbf{L}_n(I)$, the machine M is in the same state as when c has just left block $B_{i_0}^{j_0}$ in the run on $\mathbf{L}_n(J)$.

B. *Fooling argument* Let $\mathcal{V}_0, \mathcal{V}_1, \ldots$ be the sequence of states in the run of M on $\mathbf{L}_n(I)$ and let $\mathcal{W}_0, \mathcal{W}_1, \ldots$ be the sequence of states in the run of M on $\mathbf{L}_n(J)$. 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 $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$, respectively. Because of Property 3 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 $\mathbf{L}_{\text{err}}$ containing the list L_1 of $\mathbf{L}_n(I)$, the list L_2 of $\mathbf{L}_n(J)$, the lists other than L_1 and L_2 of $\mathbf{L}_n(I)$, and all reversals of the aforementioned lists. Consider M running on $\mathbf{L}_{\text{err}}$. As long as there are no cursors in block $B_{i_0}^{j_0}$ on L_1

and on L_2, the machine M running on $\mathbf{L}_{\text{err}}$ will go through the same sequence of states as on $\mathbf{L}_n(I)$. Indeed, M has not yet seen any difference between $\mathbf{L}_{\text{err}}$ on the one hand, and $\mathbf{L}_n(I)$ on the other hand. At some point, however, there may be some cursor c in block $B_{i_0}^{j_0}$.

- If this is on L_1 or $\overleftarrow{L_1}$, no cursor on L_2 or $\overleftarrow{L_2}$ will enter block $B_{i_0}^{j_0}$ as long as c is in this block (Property 1). Therefore, M will go through some successive states $\mathcal{V}_i$ (i.e., M thinks it is working on $\mathbf{L}_n(I)$) 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 3) and the machine now again goes through the same sequence of states as on $\mathbf{L}_n(I)$ (Property 2).
- If this is on L_2 or $\overleftarrow{L_2}$, we are in a similar situation: No cursor on L_1 or $\overleftarrow{L_1}$ will enter block $B_{i_0}^{j_0}$ as long as c is in this block (Property 1). Therefore, M will go through some successive states $\mathcal{W}_i$ (i.e., M thinks it is working on $\mathbf{L}_n(J)$) 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 3) and the machine now again goes through the same sequence of states as on $\mathbf{L}_n(I)$ (Property 2).

Hence, in the run of M on $\mathbf{L}_{\text{err}}$, 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 $\mathbf{L}_{\text{err}}$ finishes exactly as the run of M on $\mathbf{L}_n(I)$ after the last cursor has left block $B_{i_0}^{j_0}$. This completes the proof of Lemma 5.8. □

We can now prove:

Theorem 5.9 *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 Let M be an $o(n)$-FCM computing RST on AD-sorted inputs. Let k be the total number of cursors of M. Let $v = \binom{k}{2} + 1$ and let n be a multiple of v^2. Choose $4n$ values in $\mathbb{E}$ satisfying $a_1 < a'_1 < a_2 < a'_2 < \cdots < a_n < a'_n < b_1 < b'_1 < \cdots < b_n < b'_n$.

We fix the binary relation S of size $2n$ as follows:

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

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

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

where I^c denotes $\{1, \ldots, n\} - I$. By $\mathbf{D}(I, J)$, we denote the database consisting of the relations $R(I)$, S, and $T(J)$. It is easy to see that the nested semijoin of $R(I)$, S, and $T(J)$ is empty if, and only if, $(\pi(I) \cap J) \cup (\pi(I)^c \cap J^c) = \emptyset$. (Note that $\pi(I^c) = \pi(I)^c$.) Therefore, for each I, the query RST returns *false* on database $\mathbf{D}(I, \pi(I)^c)$,

which we will denote by $\mathbf{D}(I)$ for short. Furthermore, we observe:

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

Now, for $I \subseteq \{1, \ldots, n\}$, consider the list instance $\mathbf{L}_n(I)$ containing the lists $\mathsf{sort}_{\nearrow}\big(R(I)\big)$, $\mathsf{sort}_{\nearrow}\big(T(\pi(I)^c)\big)$, and all sorted versions of S. It is clear that the collection $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$ of these list instances is a binary (n, v)-collection with respect to (R, T). (In Definition 5.7, take $x_i = a_i$, $x'_i = a'_i$, $y_i = b_{\pi i}$, and $y'_i = b'_{\pi i}$.)

Now, we apply Lemma 5.8. We thus obtain $I, J \subseteq \{1, \ldots, n\}$ with $I \neq J$ such that the run of M on the list instance $\mathbf{L}$ containing the lists $\mathsf{sort}_{\nearrow}\big(R(I)\big)$, $\mathsf{sort}_{\nearrow}\big(T(\pi(J)^c)\big)$, all sorted versions of S, and all reversals of the aforementioned lists—in particular the lists $\mathsf{sort}_{\searrow}\big(R(I)\big)$ and $\mathsf{sort}_{\searrow}\big(T(\pi(J)^c)\big)$—ends in exactly the same way as the run of M on the list instance $\mathbf{L}_n(I)'$ containing the lists in $\mathbf{L}_n(I)$ and their reversals—in particular the lists $\mathsf{sort}_{\searrow}\big(R(I)\big)$ and $\mathsf{sort}_{\searrow}\big(T(\pi(I)^c)\big)$. Note that list instances $\mathbf{L}_n(I)'$ and $\mathbf{L}$ contain all possible sorted orders of all relations of $\mathbf{D}(I)$ and $\mathbf{D}(I, \pi(J)^c)$, respectively. Therefore, if M computes the RST query correctly on AD-sorted inputs, M returns *false* on $\mathbf{L}_n(I)'$ and *true* on $\mathbf{L}$ (cf. $(*)$). The runs of M on both list instances, however, end in the same way. We conclude that M cannot exist. □

Remark 5.10 **(a)** An analysis of the proof of Lemma 5.8 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 Lemma 5.8, 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 5.9 is sharp in terms of arity: if S would have been unary (and R and T of arbitrary arities), then the corresponding RST query would have been computable on sorted inputs.

(c) Furthermore, Theorem 5.9 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.

Using Lemma 5.8 we can also show the following strengthening of Theorem 4.3:

Theorem 5.11 *There is no* $o(n)$*-FCM working on enumerations of unary relations R and S and their reversals, that checks whether* $R \cap S \neq \emptyset$.

Proof Let M be an $o(n)$-FCM that checks whether $R \cap S \neq \emptyset$. Let k be the total number of cursors of M. Let v be $\binom{k}{2} + 1$ and let n be a multiple of v^2. Choose $2n$ pairwise distinct values $a_1, a'_1, a_2, a'_2, \ldots, a_n, a'_n$ from $\mathbb{E}$.

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

$$R(I) := \{a_\ell : \ell \in I\} \cup \{a'_\ell : \ell \in I^c\},$$
$$S(J) := \{a_\ell : \ell \in J\} \cup \{a'_\ell : \ell \in J^c\},$$

where the complements I^c and J^c are taken with respect to $\{1, \ldots, n\}$. By $\mathbf{D}(I, J)$, we denote the database consisting of the relations $R(I)$ and $S(J)$. It is easy to see that the intersection of $R(I)$ and $S(J)$ is empty if, and only if, $J = I^c$. Therefore, for each I, the intersection test fails (returns *false* on) for instance $\mathbf{D}(I, I^c)$, which we will denote by $\mathbf{D}(I)$ for short. Furthermore, we observe:

the intersection test *fails* on $\mathbf{D}(I, J)$ if, and only if, $J = I^c$. (∗∗)

Now, for $I \subseteq \{1, \ldots, n\}$, consider the list instance $\mathbf{L}_n(I)$ containing the following enumerations $R(I)_\rightarrow$ and $S(I^c)_\pi$ of $R(I)$ and $S(I^c)$, respectively: The i-th element of R is a_i if $i \in I$ and a'_i if $i \in I^c$; the $\pi_{n,v}(i)$-th element of S_π is a'_i if $i \in I$ and a_i if $i \in I^c$. (The subscripts $\rightarrow$ and π denote that the elements occur in the order of increasing indices and this latter order permuted by π, respectively.) It is clear that the collection $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$ of these list instances is a binary (n, v)-collection with respect to (R, S). (In Definition 5.7, take $x_i = y_i = a_i$, $x'_i = y'_i = a'_i$.)

Now, we apply Lemma 5.8. We thus obtain $I, J \subseteq \{1, \ldots, n\}$ with $I \neq J$ such that the run of M on the list instance $\mathbf{L}$ containing the lists $R(I)_\rightarrow$, $S(J^c)_\pi$, and their reversals ends in exactly the same way as the run of M on the list instance $\mathbf{L}_n(I)'$ containing the lists in $\mathbf{L}_n(I)$ and their reversals. If M computes the query $R \cap S \neq \emptyset$? correctly, M returns *false* on $\mathbf{L}_n(I)'$ and *true* on $\mathbf{L}$ (cf. (∗)). The runs of M on both list instances, however, end in the same way. We conclude that M cannot exist. □

Note that Theorems 5.9 and 5.11 are valid for arbitrary background structures.

6 Concluding Remarks

A natural question arising from Corollary 4.8 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. □

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 [24].

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 [8, 10] 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 [18]).

(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 $\pi_{1,2}(G)$. □

Finally, we recall the open problem already mentioned in Remark 5.5: can the semijoin $R \ltimes_{\substack{x_1<y_1 \\ x_2<y_2 \\ x_3<y_3}} S$ be computed by an FCM on AD-sorted inputs?

References

1. Aggarwal, G., Datar, M., Rajagopalan, S., Ruhl, M.: On the streaming model augmented with a sorting primitive. In: Proceedings of the 44th IEEE Symposium on Foundations of Computer Science, pp. 540–549 (2004)
2. Alon, N., Matias, Y., Szegedy, M.: The space complexity of approximating the frequency moments. J. Comput. Syst. Sci. **58**, 137–147 (1999)
3. Altinel, M., Franklin, M.: Efficient filtering of XML documents for selective dissemination of information. In: Proceedings of the 26th International Conference on Very Large Data Bases, pp. 53–64 (2000)

4. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings of the 21st ACM Symposium on Principles of Database Systems, pp. 1–16 (2002)
5. Bar-Yossef, Z., Fontoura, M., Josifovski, V.: On the memory requirements of XPath evaluation over XML streams. In: Proceedings of the 23rd ACM Symposium on Principles of Database Systems, pp. 177–188 (2004)
6. Bar-Yossef, Z., Fontoura, M., Josifovski, V.: Buffering in query evaluation over XML streams. In: Proceedings of the 24th ACM Symposium on Principles of Database Systems, pp. 216–227 (2005)
7. Chan, C.Y., Felber, P., Garofalakis, M.N., Rastogi, R.: Efficient filtering of XML documents with XPath expressions. VLDB J. **11**, 354–379 (2002)
8. Downey, R.G., Fellows, M.R.: Parameterized Complexity. Springer, Berlin (1999)
9. Fagin, R.: Degrees of acyclicity for hypergraphs and relational database schemes. J. ACM **30**, 514–550 (1983)
10. Flum, J., Grohe, M.: Parameterized Complexity Theory. Springer, Berlin (2006)
11. Garcia-Molina, H., Ullman, J.D., Widom, J.: Database System Implementation. Prentice-Hall, New York (1999)
12. Green, T.J., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata. In: Proceedings of the 9th International Conference on Database Theory, pp. 173–189 (2003)
13. Grohe, M., Koch, C., Schweikardt, N.: Tight lower bounds for query processing on streaming and external memory data. In: Proceedings of the 31st International Colloquium on Automata, Languages and Programming, pp. 1076–1088 (2005)
14. Grohe, M., Schweikardt, N.: Lower bounds for sorting with few random accesses to external memory. In: Proceedings of the 24th ACM Symposium on Principles of Database Systems, pp. 238–249 (2005)
15. Gupta, A.K., Suciu, D.: Stream processing of XPath queries with predicates. In: Proceedings of the 22th ACM SIGMOD International Conference on Management of Data, pp. 419–430 (2003)
16. Gurevich, Y.: Evolving algebras 1993: Lipari guide. In: Börger, E. (ed.) Specification and Validation Methods, pp. 9–36. Oxford University Press, London (1995)
17. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Logic **1**(1), 77–111 (2000)
18. Hella, L., Libkin, L., Nurmonen, J., Wong, L.: Logics with aggregate operators. J. ACM **48**(4), 880–907 (2001)
19. Henzinger, M., Raghavan, P., Rajagopalan, S.: Computing on data streams. External memory algorithms. In: DIMACS Ser. in Discrete Math. Theor. Comput. Sci., vol. 50, pp. 107–118. AMS, Providence (1999)
20. Hromkovič, J.: One-way multihead deterministic finite automata. Acta Inf. **19**, 377–384 (1983)
21. Law, Y.-N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Proceedings of the 30th International Conference on Very Large Data Bases, pp. 492–503 (2004)
22. Leinders, D., Van den Bussche, J.: On the complexity of division and set joins in the relational algebra. In: Proceedings of the 24th ACM Symposium on Principles of Database Systems, pp. 76–83 (2005)
23. Leinders, D., Marx, M., Tyszkiewicz, J., Van den Bussche, J.: The semijoin algebra and the guarded fragment. J. Logic Lang. Inf. **14**(3), 331–343 (2005)
24. Leinders, D., Tyszkiewicz, J., Van den Bussche, J.: On the expressive power of semijoin queries. Inf. Process. Lett. **91**(2), 93–98 (2004)
25. Libkin, L.: Elements of Finite Model Theory. Springer, Berlin (2004)
26. Muthukrishnan, S.: Data Streams: Algorithms and Applications. Now Publishers (2005)
27. Peng, F., Chawathe, S.S.: XPath queries on streaming data. In: Proceedings of the 22th ACM SIGMOD International Conference on Management of Data, pp. 431–442 (2003)
28. Rosenberg, A.L.: On multi-head finite automata. In: Proceedings of the 6th IEEE Symposium on Switching Circuit Theory and Logical Design, pp. 221–228 (1965)
29. Simmen, D., Shekita, E., Malkemus, T.: Fundamental techniques for order optimization. In: Proceedings of the 15th ACM SIGMOD International Conference on Management of Data, pp. 57–67 (1996)
30. Van den Bussche, J.: Finite cursor machines in database query processing. In: Proceedings of the 11th International Workshop on Abstract State Machines (2004)
31. Yannakakis, M.: Algorithms for acyclic database schemes. In: Proceedings of the 7th International Conference on Very Large Data Bases, pp. 82–94 (1981)