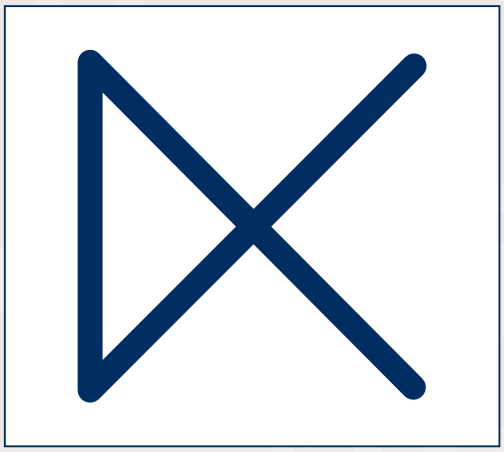
DOCTORAATSPROEFSCHRIFT

2008 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT



The Semijoin Algebra

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica, te verdedigen door:

Dirk LEINDERS

Promotor: prof. dr. Jan Van den Bussche

Universiteit Maastricht

universiteit hasselt

DOCTORAATSPROEFSCHRIFT

2008 | School voor Informatietechnologie
Kennistechnologie, Informatica, Wiskunde, ICT

The Semijoin Algebra

Proefschrift voorgelegd tot het behalen van de graad van
Doctor in de Wetenschappen: Informatica, te verdedigen door:

Dirk LEINDERS

Promotor: prof. dr. Jan Van den Bussche

Universiteit Maastricht

universiteit hasselt

D/2008/2451/10

Preface

This is the ideal opportunity to thank the people who have contributed either directly or indirectly to this thesis.

First and foremost, I would like to thank my advisor Jan Van den Bussche for his guidance and support. He was always enthusiastically prepared for discussions, during which he always managed to ask the right questions. Without Jan, this thesis would not have been possible.

I also want to thank Jerzy Tyszkiewicz, Maarten Marx, Yuri Gurevich, Nicole Schweikardt, and Martin Grohe for all the pleasant collaborations. They should receive due credit for the substantial contributions that they made to the results presented in this thesis.

Thanks to the members of the Theoretical Computer Science research group of Hasselt University for creating a stimulating environment. Among them, special thanks go out to Geert Jan Bex for being such a nice office mate during my doctoral research.

Finally, I would like to thank my family and my friends for their support. I am especially very grateful to Renee, for her patience during the past six years.

Diepenbeek, March 2008

Contents

1

Introduction

In the 1970s Codd introduced the now standard relational data model, in which a database is a finite collection of relations, where a relation is a finite set of tuples. To express queries in the relational model, Codd introduced the relational algebra (RA) with operators selection (called restriction by Codd), projection, union, difference and join [14]. Since then the relational algebra has been extensively studied [1]. A very important result is that its expressive power is equivalent to the expressive power of first-order logic, called relational calculus in database theory [15].

The "semijoin" operator, which is non-primitive in Codd's relational algebra, selects a set of tuples in one relation that have a joining tuple in another relation. The semijoin operator has also been extensively studied in the past. For example, while computing project-join queries in general is NP-complete in the size of the query and the database, this can be done in polynomial time when the database schema is acyclic [61], a property known to be equivalent to the existence of a semijoin program [11, 13, 12]. Semijoins are often used as part of a query pre-processing phase where dangling tuples are eliminated, i.e., the database is resized to the part that is relevant for answering the query. Another interesting property is that the size of a relation resulting from a semijoin is always linear in the size of the input. Therefore, a query processor will try to use semijoins as often as possible when generating a query plan for a given query (a technique known as "pushing projections" [19]). Also in distributed query processing, semijoins have great importance, because when a database is distributed across several sites, they can help avoid the shipment of many unneeded tuples.

Interestingly, to the best of our knowledge, the "semijoin algebra", which is the algebra obtained by replacing the join operator in Codd's relational algebra by the semijoin operator, was never really considered before our work.

We show that the semijoin algebra (SA) is equivalent in expressive power to the guarded fragment of first-order logic. This fragment was introduced by Andréka,

van Benthem and Németi [4] to extend modal logic from Kripke structures to arbitrary relational structures, while retaining the nice properties, such as the finite model property. Since its introduction, the guarded fragment has been studied extensively [26, 23, 25, 24, 39].

The "Codd theorem" for the semijoin algebra has a number of interesting consequences. A first consequence is that the nice properties of the guarded fragment are inherited by the semijoin algebra. One of the most important ones for database query processing is decidability. Indeed, GF has the "finite model property" [3] and is therefore decidable. Hence, the following problem, called the satisfiability problem, is decidable:

Input: An SA expression E.

Output: Is there a database D such that the result of E evaluated on D is non-empty?

As a direct consequence, we have that the equivalence problem for SA is also decidable. This means that there is an algorithm for checking whether two SA expressions always return the same result. This suggests the existence of algorithms for rewriting SA expressions into equivalent expressions that can be evaluated more efficiently. Interestingly, the satisfiability problem for first-order logic, and hence for the relational algebra and for SQL without grouping and aggregation, is not decidable.

Another consequence, related to the one above, is that using the known complexity result on the satisfiability problem for GF, we have been able to pinpoint the complexity class of the satisfiability problem for SA. The satisfiability problem for SA is EXPTIME-complete.

The Codd theorem for SA also has a number of applications. A first application is showing that a certain query can not be expressed in SA. To show that a query is not expressible in GF, there is a tool known as "guarded bisimulation". Indeed, Andréka et al. have shown that GF equals the class of first-order formulas invariant under guarded bisimulation [4].

After giving the necessary background on logic and database theory in **Chapter 2**, we discuss the Codd theorem, its consequences and the aforementioned application in **Chapter 3**.

Another application of the Codd theorem for SA is discussed in **Chapter 4**: linear query processing. Consider an RA expression E to be linear if on every input database D, the result of every subexpression of E has size linear in the size of D. In other words, when computing the result of a linear RA expression, any intermediate result has size linear in the input database. We will show that a query is expressible by a linear RA expression if and only if it is expressible by an SA expression. We will simultaneously show that an RA expression is either linear or quadratic.

This result can explain why certain operations are hard on the query processor. We elaborate on relational division and on the more general "set joins". Relational division was first identified by Codd [15] and is the prototypical example of a set join. Set joins relate database elements on the basis of sets of values, rather than single values as in a standard join. Thus, the division $R(A, B) \div S(B)$ returns all A's for which the set of B's related to A by R contains the set S. There is also a variant

Person

pName	Symptom
An	headache
An	sore throat
An	neck pain
Bob	headache
Bob	sore throat
Bob	memory loss
Bob	neck pain
Carol	headache

Disease

dName	Symptom
flu	headache
flu	sore throat
Lyme	headache
Lyme	sore throat
Lyme	memory loss
Lyme	neck pain

Symptoms

Symptom
headache
neck pain

Person $\underset{\text{Person.Symptom} \supseteq \text{Disease.Symptom}}{\bowtie}$ Disease

pName	dName
An	flu
Bob	flu
Bob	Lyme

Person $\div$ Symptoms

pName
An
Bob

Figure 1.1: An illustration of set-containment join and division.

of division, where the set of B's must equal the set S. More generally, one has the set-containment join $R \underset{B \supseteq D}{\bowtie} S$ of $R(A, B)$ and $S(C, D)$, which returns

$$\big\{(a, c) \mid \{b \mid R(a, b)\} \supseteq \{d \mid S(c, d)\}\big\},$$

and again the analogous set-equality join. In principle, any other predicate on sets could as well be used in the place of $\supseteq$ or $=$ [53, 55]. Note that a set join with predicate "intersection nonempty" boils down to an ordinary equijoin!

Example 1.1. Figure 1.1 is an illustration of the relational division operator and the set containment join. In the upper part, three relation instances Person, Disease, and Symptoms are shown. Person relates persons and symptoms; Disease relates diseases and symptoms; finally, Symptoms is a set of symptoms. The set containment join of Person and Disease on Person.Symptom $\supseteq$ Disease.Symptom is shown on the left in the lower part of Figure 1.1. The join relates a person to a disease if that person suffers all symptoms of the disease. The division of Person and Symptoms, shown on the right in the lower part, returns the persons that suffer all symptoms in relation Symptoms. □

It has long been observed that division is not well handled by classical query processing [27, 28]. Indeed, while set joins are expressible in the relational algebra using combinations of equijoins and difference operators, the resulting expressions tend to be complex and inefficient. We show in this text that division and (an emptiness test for) set containment/equality join can not be expressed in SA. Therefore, any RA expression for these operators must produce intermediate results of quadratic size. Our work thus provides a formal justification of work done by various authors on implementing set joins directly as special-purpose operators, or on implementing them by compiling to the more powerful version of the relational algebra that includes

grouping, sorting, and aggregation operators [37, 48, 52]. For instance, division (and set-equality join) can be implemented efficiently in time $O(n \log n)$ using sorting or counting tricks.[1]

Processing of semijoin algebra expressions can thus be accomplished using linear space, but how many passes through the data do we need to compute the answer to a semijoin algebra expression? We answer this question in **Chapter 5**. In database query processing, one-pass and two-pass algorithms are distinguished [19]. One-pass algorithms read the data only once from disk. Two-pass algorithms read the data from disk a first time, process the data in some way, write the data to disk, and finally read the data a second time to further process it. Processing of the data after the first scan usually includes sorting or hashing. It is clear that the selection operator only requires a single pass through the data, as each tuple in the database can be processed independently. When duplicate elimination is abandoned, also the projection and the union operator can be implemented by doing a single pass. The difference and the semijoin of two relations intuitively can not be computed in a single pass.

To prove this formally we introduce a model called finite cursor machines (FCMs). A finite cursor machine works on a number of lists of tuples and can operate in a finite number of modes using an internal memory in which it can store bit strings. An FCM accesses each relation through finitely many cursors, each of which can read one tuple of a list at any time. A list of tuples can be produced as output. The model incorporates certain "streaming" or "sequential processing" aspects by imposing two restrictions: First, the cursors can only move on the lists sequentially in one direction. Thus once the last cursor has left a tuple of a list, this tuple can never be accessed again during the computation. Second, the internal memory is limited. The model is clearly inspired by the abstract state machine (ASM) methodology [31, 32], and indeed we will formally define our model using this methodology.

We prove that the semijoin and difference operation can not be computed by an FCM, not even when the FCM is allowed to store bit strings of size $o(n)$, where n is the length of the input lists. When on all sorted versions of the database relations are provided as input, however, every operator of the semijoin algebra can be computed by an FCM. Consequently, every query in the semijoin algebra can be computed by a query plan composed of finite cursor machines and sorting operations. In such query plans, in general, a lot of intermediate sorting operations are introduced. 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 [56]. 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: 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. We simultaneously show that, while FCMs working on both ascendingly and descendingly sorted inputs are strictly more powerful — i.e., they can compute more relational algebra queries — than FCMs working on ascendingly sorted inputs only, descending sorting can not help in avoiding intermediate sorting.

[1] For set-equality join, where the result size alone can already be quadratic, we should really say in time $O(n \log n)$ plus output size.

From the streaming aspects of finite cursor machines, in **Chapter 6** we move on to the topic of stream query processing, which has received a lot of attention in the database systems research community over the past few years. We give just a few references here [57, 9, 10, 47, 20]; much more has been published. Stream queries are typically "continuous" in that their result must be continually updated as new data arrives: indeed, stream applications are "data-driven". Consequently, continuous stream queries must be computed in an incremental fashion, using so-called "non-blocking" operators. Relational algebra operators that are monotone are non-blocking; query operators that are not monotone, such as difference, or grouping and aggregation, are typically made non-blocking by restricting them to sliding windows.

We first offer a theoretical framework that attempts to clarify various philosophical questions about stream queries. For instance, if streams are thought of as infinite, and arbitrary queries are modeled as functions from streams to streams, what does it mean for a query to be computable? Is computability the same concept as continuity? What is the precise connection between continuity and monotonicity? Can one give a formal definition of what it means for an arbitrary operator to be non-blocking?

Earlier work in this direction has already been reported by Arasu and Widom [7] and by Law, Wang and Zaniolo [41]. Our work has the following new features.

First, we distinguish from the outset between timed and untimed applications. In a timed setting, the timestamps in the output stream of some stream query are synchronized with the timestamps in the input stream; in an untimed setting, they are not. The usual applications mentioned in the data stream literature, such as stock quotes or sensors, are timed. Nevertheless, untimed streams also find applications, e.g., in audio or video streams, or Internet broadcasts, where the logical order among arriving packets is more important than precise timing information. More fundamentally, however, much of the theory of stream queries can already be developed on the more basic untimed level, viewing timed streams merely as a special case of untimed streams. Nonetheless, we will also identify some specific aspects of timed queries, in particular, their non-predicting nature (in a sense that will be made precise later).

Second, our formal definitions of abstract computable stream queries are grounded in the theory of type-2 effectivity (TTE) [60]. This is a well-established theory of computability on infinite strings (and much more, which we will not use here). The basic idea of TTE, strikingly analogous to the idea of continuous stream queries, is that arbitrary long finite prefixes of the infinite output can be computed from longer and longer finite prefixes of the infinite input. A basic insight from TTE is that computable functions on infinite strings are indeed "continuous", but now in the precise sense of mathematical topology. More specifically, under a natural metric on infinite strings (known as the Cantor metric), where two strings are closer the longer they agree on their prefixes, computable functions can be shown to be continuous in the standard mathematical sense of the word. Continuity is a useful property for it provides us with a principled way to prove that not just any function from streams to streams can be naturally considered to be a stream query.

And finally, our theory is abstract in the sense that elements from a stream can come from an arbitrary universe, equipped with predicates and functions. In mathematical logic one speaks of a structure, and we will refer to the universe as the background structure. In particular, we do not concern ourselves with the encoding

of stream elements as bit strings (finite or infinite), or with Turing machine computations on those bit strings, since those aspects are already well understood from the TTE. Consequently, our theory is very general, and computable stream queries will turn out to be the same thing as continuous functions from streams to streams (where we introduce a variant of the Cantor topology that accommodates finite as well as infinite streams).

We will argue that finite cursor machines are unrealistically powerful in the streaming context. We therefore introduce a new model, again based on the Abstract State Machine methodology [31, 32]. We call our model "streaming ASM". We show that every computable stream query is computable by a streaming ASM with an appropriate background structure. Moreover, streaming ASMs allow us to prove impossibility results. Specifically, we focus on bounded memory machines: such machines can only remember a constant number of previously seen stream elements. Bounded memory machines are natural in the context of query processing; for example, any query operator that applies a sliding window (typical in streaming applications) is computable in bounded memory. We will prove that there exist simple queries that are not bounded memory computable, one of the simplest being the query INTERSECT: finding the common elements in two interleaved streams.

As FCMs are certainly at least as powerful as streaming ASMs and since we already know that the query INTERSECT mentioned above is not even computable by an FCM, it follows by a reduction that the query is also not computable by a streaming ASM. Yet, we will give a direct proof of this result, that is much simpler and thus provides more direct insight on the limitations of bounded memory stream processing. Moreover, we will see that there exist stream queries that are computable by an FCM, but not by a streaming ASM.

Finally, in **Chapter 7** we study the expressive power of the semijoin algebra in the presence of arbitrary predicates in the selection and join conditions. Note that the Codd theorem for the semijoin algebra in Chapter 3 only considers equi-semijoins.

The first part of the study deals with repetitions and permutations of columns. The projection operator of Codd's relational algebra can permute and repeat columns. This permuting and repeating of columns, however, does not add expressive power to the relational algebra. Indeed, the two existing perspectives on the relational model, namely the named perspective, in which tuples are viewed as functions from the set of attributes to the domain, and the unnamed perspective, in which tuples are viewed as ordered lists of domain values, are equivalent [1], whereas permuting and repeating of columns can not be done in the named perspective. For completeness, we will explicitly show that any relational algebra expression can be rewritten into an equivalent relational algebra expression where no projection operator permutes or repeats columns.

While in the full relational algebra permuting and repeating of columns does not add expressive power, this is not clear for the semijoin algebra. Indeed, the rewrite rule to replace a permuting or repeating projection in a relational algebra expression with a non-permuting and non-repeating one uses the join operator, that the semijoin algebra lacks. Nevertheless, we show that any semijoin algebra expression can still be simulated by semijoin algebra expressions where no projection operator permutes or repeats columns. The notion of "simulation", however, becomes more complicated.

The idea is that given an arbitrary expression E, one can produce a set of permutation- and repetition-free expressions that return the relevant values of the output tuples of E, up to certain repetitions and permutations which are produced as a by-product of the translation. In particular, for boolean expressions, there is always a single equivalent boolean expression that is permutation- and repetition-free.

In a second part of the study, we define an Ehrenfeucht-Fraïssé game, that characterizes the discerning power of the semijoin algebra in the presence of arbitrary predicates in the selection and join conditions. Using the Ehrenfeucht-Fraïssé game as a tool, we will particularly study the expressive power of $\mathrm{SA}^{\neq}$ and $\mathrm{SA}^{<,<}$. The selection and join conditions in these algebras are quantifier-free formulas over the vocabulary $\{=\}$ and $\{=,<\}$, respectively.

2

Preliminaries

In this chapter we give the necessary background on logic and database theory.

From the outset, we assume a universe $\mathbb{U}$ of basic data elements. Over this universe, various predicates are defined. The names of these predicates and their arities are collected in the vocabulary Ω. The equality predicate $(=)$ is always in Ω. Furthermore, we assume the existence of two disjoint infinite sets of variables $X = \{x_1, x_2, \ldots\}$ and $Y = \{y_1, y_2, \ldots\}$. We will use V to denote $X \uplus Y$.

We recall:

Definition 2.1 (conjunctive formula, quantifier-free formula)**.** An *atomic formula* over Ω is either a formula of the form $v_1 = v_2$, or a formula of the form $p(v_1, \ldots, v_n)$, where $v_1, \ldots, v_n$ are variables in V and p is a predicate in Ω.

All atomic formulas over Ω are *conjunctive formulas* over Ω. If φ and ψ are conjunctive formulas, then so is $\varphi \wedge \psi$. All atomic formulas over Ω are *quantifier-free formulas* over Ω. If φ and ψ are quantifier-free formulas, then so are $\neg\varphi$ and $\varphi \wedge \psi$.

The set of all conjunctive formulas over Ω is denoted by $\mathrm{cf}(\Omega)$; the set of all quantifier-free formulas over Ω is denoted by $\mathrm{qff}(\Omega)$.

Throughout the text, we fix an arbitrary database schema $\mathbf{S}$. A database schema is a finite set of relation names, where each relation name R in $\mathbf{S}$ has an associated arity, denoted by $\mathrm{arity}(R)$. A relation instance of R is a finite subset of $\mathbb{U}^n$, where n is the arity of R. Elements of a relation instance are often called tuples, or rows. A database D over $\mathbf{S}$ is an assignment of a relation instance $D(R)$ to each $R \in \mathbf{S}$. By $adom(D)$, we denote the set of elements in $\mathbb{U}$ occurring in D; we will refer to this set as the active domain of D. Database names will be denoted by upper case letters. We will use D when only a single database is the object of case; in other cases we will use A, B, ... A *query* is a mapping Q from databases to relations, such that the relation $Q(D)$ is the answer to the query Q on database D.

A *list instance* of R is a finite list of n-tuples, i.e., elements of $\mathbb{U}^n$, where n is the arity of relation name R in $\mathbf{S}$. A *list database* with schema $\mathbf{S}$ assigns a list instance to each relation name $R \in \mathbf{S}$.

2.1 First-order logic and its guarded fragment

We now recall the definition of first-order logic and of its "guarded" fragment, which we will use in this text.

Definition 2.2 (First-order logic (FO))**.** Formulas in FO are defined inductively as follows.

1. Every quantifier-free formula over $\mathbf{S}$ is a formula in FO.
2. If φ and ψ are formulas in FO, then so are $\neg\varphi$ and $\varphi \wedge \psi$.
3. If φ is in FO and v is a variable in V, then also $\exists v\varphi$ is a formula in FO.

An occurrence of a variable v in a formula φ is *free* if φ is an atomic formula; or if $\varphi = \psi \wedge \xi$ and the occurrence of v is free in ψ or in ξ; or if $\varphi = \exists y\psi$ and the occurrence of v is free in ψ. An occurrence of v in φ is *bound* if it is not free. The set of *free variables* in φ, denoted $\mathit{free}(\varphi)$, is the set of all variables that have at least one free occurrence in φ.

The semantics of an FO formula interpreted on a database D is defined in terms of valuations over the set of free variables. A *valuation* over $\mathit{free}(\varphi)$ is a total function ν from $\mathit{free}(\varphi)$ to $\mathit{adom}(D)$. Database D satisfies φ under ν, denoted $D \models \varphi[\nu]$, if

- $\varphi = R(\overline{v})$ and $\nu(\overline{v}) \in D(R)$; or
- $\varphi = \psi \wedge \xi$ and $D \models \psi[\nu|_{\mathit{free}(\psi)}]$ and $D \models \psi[\nu|_{\mathit{free}(\xi)}]$; or
- $\varphi = \neg\psi$ and $D \not\models \psi[\nu]$; or
- $\varphi = \exists v\psi$ and for some $d \in \mathit{adom}(D)$, we have $D \models \psi[\nu \cup \{v \to d\}]$.

Here, $\nu \cup \{v \to d\}$ denotes the valuation with domain $\mathbf{dom}(\nu) \cup \{v\}$ that is identical to ν and maps v to d.

Proviso. When φ stands for a first-order formula, then we will write $\varphi(v_1, \ldots, v_n)$ to indicate that all free variables of φ are among $v_1, \ldots, v_n$, i.e., $\mathit{free}(\varphi) \subseteq \{v_1, \ldots, v_n\}$. Also, for a tuple $(a_1, \ldots, a_n)$ of elements in $\mathbb{U}$, we will write $\varphi(\overline{a})$ to denote the truth value of φ under the valuation ν that maps each free variable v_i to element a_i.

Definition 2.3 (Guarded fragment (GF))**.** Formulas in GF are inductively defined as follows.

1. Every quantifier-free formula over $\mathbf{S}$ is a formula in GF.
2. If φ and ψ are formulas in GF, then so are $\neg\varphi$ and $\varphi \wedge \psi$.

3. Let $\varphi(\overline{x},\overline{y})$ be a formula in GF and let $\alpha(\overline{x},\overline{y})$ be a relation atom over $\mathbf{S}$ (i.e., an atomic formula $R(\ldots)$ with $R \in \mathbf{S}$). If all free variables of φ do actually occur in α then $\exists\overline{y}(\alpha(\overline{x},\overline{y}) \wedge \varphi(\overline{x},\overline{y}))$ is a formula in GF.

As the guarded fragment is a fragment of first-order logic, the semantics of GF is that of first-order logic.

An FO (GF) query is an expression of the form $\{e_1,\ldots,e_n \mid \varphi\}$ where φ is a formula in FO (GF) and $e_1,\ldots,e_n$ are variables such that $\{e_1,\ldots,e_n\} = \mathit{free}(\varphi)$. Let Q be an FO (GF) query and let D be a database. The semantics of Q on D, denoted $Q(D)$, is

$$Q(D) = \{(\nu(e_1),\ldots,\nu(e_n)) \mid D \models \varphi[\nu] \text{ and } \nu \text{ is a valuation over } \mathit{free}(\varphi)\}.$$

Example 2.4. Suppose $\mathbf{S}$ is Ullman's well-known example schema [58]

$$\{\text{Likes(drinker,beer), Serves(bar,beer), Visits(drinker,bar)}\}.$$

Let us call a bar lousy if it only serves beers nobody likes. The query that asks for the lousy bars can be expressed in GF as follows:

$$\{x \mid \exists y\,\text{Visits}(x,y) \wedge \neg\exists x\,\big(\text{Serves}(y,x) \wedge \exists y\,\text{Likes}(y,x)\big)\}. \quad \square$$

The guarded fragment has been studied extensively [4, 23, 25, 16]. An important result is the invariance under an equivalence relation on databases, known as "guarded bisimilarity". Before we define guarded bisimilarity and state the invariance property of the guarded fragment, we need to introduce the notions of "guarded set" and "guarded tuple".

Definition 2.5 (guarded set, guarded tuple)**.** Let D be a database over schema $\mathbf{S}$.

- A set $X \subseteq \mathbb{U}$ is guarded in D if there exists a tuple $(a_1,\ldots,a_k) \in D(R)$ (for some R in $\mathbf{S}$) such that $X = \{a_1,\ldots,a_k\}$.
- A tuple $(a_1,\ldots,a_n) \in \mathbb{U}^n$ is guarded in D if $\{a_1,\ldots,a_n\} \subseteq X$ for some guarded set X in D.

For completeness, we recall the definition of partial isomorphism.

Definition 2.6 (partial isomorphism)**.** Let A and B be two databases over schema $\mathbf{S}$. For $X, Y \subseteq \mathbb{U}$, a mapping $f : X \to Y$ is a *partial isomorphism* from A to B if it is bijective, and for each $R \in \mathbf{S}$, of arity n, and all $x_1,\ldots,x_n \in X$, we have $(x_1,\ldots,x_n) \in A(R) \Leftrightarrow (f(x_1),\ldots,f(x_n)) \in B(R)$.

Definition 2.7 (guarded bisimulation, guarded bisimilarity)**.** A *guarded bisimulation* between two databases A and B is a non-empty set $\mathcal{I}$ of finite partial isomorphisms from A to B, such that the following back and forth conditions are satisfied:

Forth. For every $f\colon X \to Y$ in $\mathcal{I}$ and for every guarded set X', there exists a partial isomorphism $g\colon X' \to Y'$ in $\mathcal{I}$ such that f and g agree on $X \cap X'$.

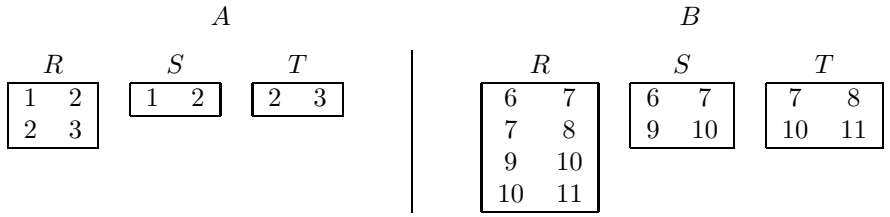


Figure 2.1: Databases A and B to illustrate the notion of guarded bisimulation.

Back. For every $f\colon X \to Y$ in $\mathcal{I}$ and for every guarded set Y', there exists a partial isomorphism $g\colon X' \to Y'$ in $\mathcal{I}$ such that f^{-1} and g^{-1} agree on $Y \cap Y'$.

Now let A be a database and $\overline{a}$ a guarded tuple in A, and let $B, \overline{b}$ be another such pair. We say that $A, \overline{a}$ and $B, \overline{b}$ are *guarded bisimilar*—denoted by $A, \overline{a} \sim_g B, \overline{b}$—if there exists a guarded bisimulation $\mathcal{I}$ between them that contains $\overline{a} \mapsto \overline{b}$.

We illustrate the notion of guarded bisimilarity with an example.

Example 2.8. Let A and B be the databases shown in Figure 2.1. The following set $\mathcal{I}$ of partial isomorphisms is a guarded bisimulation between A and B:

$$
\begin{array}{ll}
1 \mapsto 6 & \\
(1,2) \mapsto (6,7) & (2,3) \mapsto (7,8) \\
(1,2) \mapsto (9,10) & (2,3) \mapsto (10,11)
\end{array}
$$

Let us check the back property for one particular partial isomorphism $f\colon (1,2) \mapsto (6,7)$. We consider all guarded sets Y' of B: if Y' is $(6,7)$, we choose g as f; if Y' is $(9,10)$, we also choose g as f (the intersection of Y and Y' is empty, so any g will do); if Y' is $(7,8)$, we choose $(2,3) \mapsto (7,8)$ for g (the intersection of Y and Y' is $\{7\}$ and f^{-1} and g^{-1} both map 7 to 2); finally, if Y' is $(10,11)$, we choose $(2,3) \mapsto (10,11)$ for g (the intersection of Y and Y' is $\{10\}$ and f^{-1} and g^{-1} both map 10 to 2). The other properties can be checked analogously. We thus have $A, 1 \sim_g B, 6$. □

We can now recall [4]:

Proposition 2.9 (Andréka et al. [4])**.** *An FO formula φ is invariant under guarded bisimulations if and only if φ is in GF.*

2.2 The relational algebra and the semijoin algebra

We define the relational algebra (RA) and the semijoin algebra (SA) parameterized by the allowed selection and (semi)join conditions. Let Ω_σ, $\Omega_\bowtie$, and $\Omega_\ltimes$ be subsets of Ω. The predicates in Ω_σ are called selection predicates; the predicates in $\Omega_\bowtie$ are called join predicates; and the predicates in $\Omega_\ltimes$ are called semijoin predicates. Furthermore, let Φ_σ, $\Phi_\bowtie$, and $\Phi_\ltimes$ be subsets of $\mathrm{qff}(\Omega_\sigma)$, $\mathrm{qff}(\Omega_\bowtie)$, and $\mathrm{qff}(\Omega_\ltimes)$ respectively. The sets Φ_σ, $\Phi_\bowtie$, and $\Phi_\ltimes$ define the selection conditions, join conditions, and semijoin conditions.

Definition 2.10 (relational algebra, RA)**.** The syntax and semantics of the relational algebra are inductively defined as follows:

1. Each relation name $R \in \mathbf{S}$ is a relational algebra expression. Its arity comes from $\mathbf{S}$.
2. If $E_1, E_2 \in$ RA have arity n, then also $E_1 \cup E_2$ (union), $E_1 - E_2$ (difference) belong to RA and are of arity n.
3. If $E \in$ RA has arity n and $i_1, \ldots, i_k \in \{1, \ldots, n\}$, then $\pi_{i_1,\ldots,i_k}(E)$ (projection) belongs to RA and is of arity k.
4. If $E \in$ RA has arity n and $\theta(x_1, \ldots, x_n)$ is a condition in Φ_σ, then $\sigma_\theta(E)$ (selection) belongs to RA and is of arity n.
5. Let $E_1, E_2 \in$ RA with arities n and m, respectively. If $\theta(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a condition in $\Phi_\bowtie$, then $E_1 \bowtie_\theta E_2$ (join) belongs to RA and is of arity $n+m$.

Let E be an RA expression and let D be a database. Then the result of E on D, denoted $E(D)$, is defined inductively as follows:

1. $R(D) := D(R)$.
2. $E_1 \cup E_2(D) := E_1(D) \cup E_2(D)$, $E_1 - E_2(D) := E_1(D) - E_2(D)$.
3. $\pi_{i_1,\ldots,i_k} E(D) := \{(a_{i_1}, \ldots, a_{i_k}) \mid (a_1, \ldots, a_n) \in E(D)\}$.
4. $\sigma_\theta E(D) := \{\overline{a} \in E(D) \mid \theta(\overline{a}) \text{ is true}\}$.
5. $E_1 \bowtie_\theta E_2(D) := \{(\overline{a}, \overline{b}) \mid \overline{a} \in E_1(D), \overline{b} \in E_2(D) : \theta(\overline{a}, \overline{b}) \text{ is true}\}$.

Codd [14, 15] has shown that every FO query can be expressed as an RA query and vice versa. We recall the result here as a theorem:

Theorem 2.11 (Codd [14, 15])**.** *The class of first-order queries and the class of relational algebra queries coincide.*

Definition 2.12 (semijoin algebra, SA)**.** The semijoin algebra is the variant of RA obtained by replacing the join operator $E_1 \bowtie_\theta E_2$ by the semijoin operator $E_1 \ltimes_\theta E_2$. The semantics of the semijoin operator is as follows:

$$E_1 \ltimes_\theta E_2(D) := \{\overline{a} \in E_1(D) \mid \exists \overline{b} \in E_2(D) : \theta(\overline{a}, \overline{b}) \text{ is true}\}$$

Notation. To refer to RA with selection and join conditions Φ_σ and $\Phi_\bowtie$ respectively, we will use the notation RA[Φ_σ, $\Phi_\bowtie$] and similarly for SA[Φ_σ, $\Phi_\ltimes$]. Furthermore, we use the following abbreviations:

$$\begin{aligned}
\mathrm{RA} &= \mathrm{RA}[\mathrm{qff}(=), \mathrm{cf}(=)] \\
\mathrm{SA} &= \mathrm{SA}[\mathrm{qff}(=), \mathrm{cf}(=)] \\
\mathrm{SA}^{\neq} &= \mathrm{SA}[\mathrm{qff}(=), \mathrm{qff}(=)] \\
\mathrm{RA}^{<,<} &= \mathrm{RA}[\mathrm{qff}(=,<), \mathrm{qff}(=,<)] \\
\mathrm{SA}^{<,<} &= \mathrm{SA}[\mathrm{qff}(=,<), \mathrm{qff}(=,<)] \\
\mathrm{SA}^{<,=} &= \mathrm{SA}[\mathrm{qff}(=,<), \mathrm{cf}(=)].
\end{aligned}$$

Example 2.13. Suppose again the beer-drinkers database schema of Example 2.4. The query that asks for the visitors of lousy bars can be expressed in SA as follows:

$$E := \big(\pi_2(\text{Visits}) - \pi_1(\text{Serves})\big) \cup \big(\pi_1(\text{Serves}) - \pi_1(\text{Serves} \underset{2=2}{\ltimes} \text{Likes})\big).$$

The left operand of the union returns the bars that do not serve any beers; the right operand of the union returns the bars that serve at least one beer, but that do not serve a beer somebody likes.

The query that asks for the drinkers that visit a lousy bar can be expressed in SA as follows:

$$\pi_1\big(\text{Visits} \underset{2=1}{\ltimes} E\big).$$

□

Note that in the example above, we have only written the subscripts in the semijoin condition $x_2 = y_1$, i.e., $2 = 1$. We will often do so for selection and join conditions as well.

3

A Codd theorem for the semijoin algebra

In this chapter, we show a completeness theorem for the semijoin algebra. In particular, we show that the class of semijoin algebra queries and the class of guarded fragment queries coincide.

3.1 Motivating example

Relational algebra expressions are built up from relation names in the database schema using the operations selection, projection, union, difference, and join. Semijoin algebra expressions are built up in the same way, using the same set of operations, except for the join. Instead of the join operation, the semijoin operation is available. Intuitively, the semijoin operation is less powerful than the join operation: the semijoin operation can not *produce* combinations of tuples from different relations. *Checking* whether a join is empty or not can be accomplished, however, using a semijoin. Indeed, for all relations R and S and for all conditions θ, we have that $R \bowtie_\theta S \neq \varnothing$ if and only if $R \ltimes_\theta S \neq \varnothing$. Checking whether an arbitrary join expression—i.e., an expression with only relation names and join operators—is empty or not, however, is outside the capabilities of the semijoin operation: there is a join expression $E_\bowtie$ for which there is no semijoin expression $E_\ltimes$ such that $E_\bowtie(D) \neq \varnothing$ iff $E_\ltimes(D) \neq \varnothing$, for every database D. In fact, the join expressions $E_\bowtie$ for which there exists such a semijoin expression $E_\ltimes$ have been characterized by "acyclic join expressions" [11, 12, 13].

Example 3.1. Consider the beer-drinkers database schema from Example 2.13. The boolean query Q:

Is there a drinker that visits a bar that serves a beer he likes?

will return true on a database if and only if the result of the following relational algebra expression is nonempty:

$$(\text{Serves} \underset{2=2}{\ltimes} \text{Likes}) \underset{\substack{3=1\\1=2}}{\ltimes} \text{Visits}.$$

The query Q is an example of a cyclic join query and hence, there is no semijoin expression for Q. □

Note that semijoin expressions can use *only* semijoins, while semijoin algebra expressions can also use selection, projection, union, and difference. It is therefore still an interesting question whether the inclusion SA $\subseteq$ RA is strict. The intuition is SA $\subsetneq$ RA. In this chapter we confirm this intuition. Before being able to do so, we more thoroughly study the expressive power of SA. Motivated by Codd's completeness theorem (see Theorem 2.11), we ask ourselves the question: "To which fragment of FO is SA equivalent?" We thus want to fill in the lower left question mark in the following diagram:

$$\begin{array}{ccc} \text{SA} & \overset{?}{\subsetneq} & \text{RA} \\ \| & & \| \\ ? & \overset{?}{\subsetneq} & \text{FO} \end{array}$$

The fragment of FO that SA is equivalent to will turn out to be the well-known guarded fragment. Hence it will follow that both inclusions in the above diagram are strict.

3.2 Codd theorem for SA

Before we prove that SA is subsumed by GF, we need a lemma that says that each tuple in the result of an SA expression E on a database D is guarded. For the definitions of "guarded set" and "guarded tuple", see Chapter 2.

Lemma 3.2. *For every SA expression E, for every database D over $\mathbf{S}$, and for every tuple $\overline{a}$ in $E(D)$, we have that $\overline{a}$ is guarded.*

Proof. By structural induction on expression E. □

The set of guarded k-tuples in databases over schema $\mathbf{S} = \{R_1, \ldots, R_t\}$ can be defined by the following formula [25]:

$$\mathbb{G}_k(x_1, \ldots, x_k) := \bigvee_{i=1}^{t} \exists \overline{y} \left(R_i \overline{y} \wedge \bigwedge_{l=1}^{k} \bigvee_{j} x_l = y_j \right)$$

Note that this formula is syntactically *not* in GF. Nevertheless, it can be equivalently expressed in GF as follows. For any complete equality type on $\{x_1, \ldots, x_k\}$ specified by a quantifier-free formula $\eta(\overline{x})$ in the language of just $=$, let $\overline{x}^\eta$ be a subtuple of $\overline{x}$ comprising precisely one variable from each $=$-class specified by η. Let $\alpha(\overline{x}^\eta, \overline{y})$ be an

atomic formula over $\mathbf{S}$ in which all variables in $\overline{x}^\eta$ actually occur and the $\overline{y}$ are new, i.e., disjoint from $\overline{x}$. It is clear that the formula

$$\bigvee_\eta \bigvee_\alpha \eta(\overline{x}) \wedge \exists \overline{y} \alpha(\overline{x}^\eta, \overline{y})$$

is in GF and is equivalent to $\mathbb{G}_k(x_1, \ldots, x_k)$. The following lemma is now clear. It will also be of use in the proof of Theorem 3.4.

Lemma 3.3. *If $\varphi(\overline{x}, \overline{y})$ is in GF, then $\exists \overline{y}(\mathbb{G}(\overline{x}, \overline{y}) \wedge \varphi(\overline{x}, \overline{y}))$ and $\forall \overline{y}(\mathbb{G}(\overline{x}, \overline{y}) \rightarrow \varphi(\overline{x}, \overline{y}))$ can be equivalently expressed in GF.*

This lemma implies that, if we regard $\mathbb{G}_m$ as a relation symbol, with m the maximal arity of relation symbols in $\mathbf{S}$, each GF sentence is equivalent to a sentence of the guarded logic where we always use $\mathbb{G}_m$ as the guard. It is interesting to note that historically, GF has its roots in relativized cylindric algebras, where we indeed relativize all operations to a single relation [34, 51, 50].

Proviso. In the rest of this chapter, we use m to denote the maximal arity of the relation names in $\mathbf{S}$.

We now prove that SA is subsumed by GF.

Theorem 3.4. *For every SA expression E of arity k, there exists a GF formula $\varphi_E(x_1, \ldots, x_k)$ such that for every database D over $\mathbf{S}$ and for every tuple $\overline{a}$ in $\mathbb{U}^k$, we have $\overline{a} \in E(D)$ iff $D \models \varphi_E(\overline{a})$.*

Proof. The proof is by structural induction on E.

- if E is R, then $\varphi_E(x_1, \ldots, x_k) := R(x_1, \ldots, x_k)$.
- if E is $E_1 \cup E_2$, then

$$\varphi_E(x_1, \ldots, x_k) := \varphi_{E_1}(x_1, \ldots, x_k) \vee \varphi_{E_2}(x_1, \ldots, x_k).$$

- if E is $E_1 - E_2$, then

$$\varphi_E(x_1, \ldots, x_k) := \varphi_{E_1}(x_1, \ldots, x_k) \wedge \neg \varphi_{E_2}(x_1, \ldots, x_k).$$

- if E is $\sigma_{i=j}(E_1)$, then $\varphi_E(x_1, \ldots, x_k) := \varphi_{E_1}(x_1, \ldots, x_k) \wedge x_i = x_j$.
- if E is $\pi_{i_1, \ldots, i_k}(E_1)$ with E_1 of arity n, then, by induction, we have a formula $\varphi_{E_1}(z_1, \ldots, z_n)$. Now replace in $\varphi_{E_1}(\overline{z})$, for $j = 1, \ldots, k$, each occurrence of z_{i_j} by x_j, and replace, for $l \notin \{i_j \mid j = 1, \ldots, k\}$, each occurrence of z_l by y_l. Let the resulting formula be $\psi(\overline{x}, \overline{y})$. By Lemma 3.2, $\psi(\overline{x}, \overline{y})$ is equivalent to the formula $\mathbb{G}_n(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y})$. Now, $\varphi_E(x_1, \ldots, x_k)$ is the formula

$$\exists \overline{y}(\mathbb{G}_n(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y}))$$

  which can be written guarded by Lemma 3.3.

- if E is $E_1 \ltimes_\theta E_2$ with $\theta = \bigwedge_{l=1}^{s} x_{i_l} = y_{j_l}$ and E_2 of arity n, then, by induction, we have formulas $\varphi_{E_1}(x_1, \ldots, x_k)$ and $\varphi_{E_2}(z_1, \ldots, z_n)$. Now replace in $\varphi_{E_2}(\overline{z})$, for $l = 1, \ldots, s$, each occurrence of z_{j_l} by x_{i_l}, and replace, for $i \notin \{j_l \mid l = 1, \ldots, s\}$, each occurrence of z_i by y_i. Let the resulting formula be $\psi(\overline{x}, \overline{y})$. By Lemma 3.2, $\psi(\overline{x}, \overline{y})$ is equivalent to the formula $\mathbb{G}_n(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y})$. Now, $\varphi_E(x_1, \ldots, x_k)$ is the formula
$$\varphi_{E_1}(x_1, \ldots, x_k) \wedge \exists \overline{y}(\mathbb{G}_n(\overline{x}, \overline{y}) \wedge \psi(\overline{x}, \overline{y}))$$
which can be written guarded by Lemma 3.3. Note that condition θ is enforced by repetition of variables x_{i_l}. □

The literal converse statement of Theorem 3.4 is not true, because the guarded fragment contains all quantifier-free first-order formulas, so that one can express arbitrary Cartesian products in it, such as $\{(x, y) \mid S_1(x) \wedge S_2(y)\}$. Cartesian products, of course, can not be expressed in the semijoin algebra. Nevertheless, the result of any GF query restricted to guarded k-tuples, where $k \leqslant m$, is always expressible in SA.

It is clear that for every database D over $\mathbf{S}$ and for every $k \leqslant m$, the set of guarded k-tuples in D equals $G_k(D)$, where G_k is the SA expression
$$\bigcup_{R \in \mathbf{S}} \{\pi_{i_1, \ldots, i_k} R \mid 1 \leqslant i_1, \ldots, i_k \leqslant \text{arity}(R)\}.$$

We now prove

Theorem 3.5. *For every GF formula $\varphi(x_1, \ldots, x_k)$ with $k \leqslant m$, there exists an SA expression $E_\varphi^{(x_1, \ldots, x_k)}$ such that for every database D and for every guarded tuple $\overline{a}$ in D, we have $D \models \varphi(\overline{a})$ iff $\overline{a} \in E_\varphi^{(x_1, \ldots, x_k)}(D)$.*

Proof. By structural induction on φ, we construct the desired semijoin expression $E_\varphi^{(x_1, \ldots, x_k)}$.

- if $\varphi(x_1, \ldots, x_k)$ is $R(x_{i_1}, \ldots, x_{i_l})$ then $E_\varphi^{(x_1, \ldots, x_k)} := G_k \ltimes_\theta R$, where θ is $(x_{i_1} = y_1) \wedge (x_{i_2} = y_2) \wedge \ldots \wedge (x_{i_l} = y_l)$;
- if $\varphi(x_1, \ldots, x_k)$ is $(x_i = x_j)$ then $E_\varphi^{(x_1, \ldots, x_k)} := \sigma_{i=j}(G_k)$;
- if $\varphi(x_1, \ldots, x_k)$ is $\psi(x_1, \ldots, x_k) \vee \xi(x_1, \ldots, x_k)$ then $E_\varphi^{(x_1, \ldots, x_k)} := E_\psi^{(x_1, \ldots, x_k)} \cup E_\xi^{(x_1, \ldots, x_k)}$;
- if $\varphi(x_1, \ldots, x_k)$ is $\neg\psi(x_1, \ldots, x_k)$ then $E_\varphi^{(x_1, \ldots, x_k)} := G_k - E_\psi^{(x_1, \ldots, x_k)}$;
- suppose $\varphi(x_1, \ldots, x_k)$ is $\exists z_1, \ldots, z_p(\alpha(\overline{x}, \overline{z}) \wedge \psi(\overline{x}, \overline{z}))$. First, note that not every x_i may effectively occur in α. So, let $x_{i_1}, \ldots, x_{i_r}$ be the variables among $x_1, \ldots, x_k$ that effectively occur in α. By induction, we have expressions
$$E_\alpha^{(x_{i_1}, \ldots, x_{i_r}, z_1, \ldots, z_p)} \quad \text{and} \quad E_\psi^{(x_{i_1}, \ldots, x_{i_r}, z_1, \ldots, z_p)}.$$

Note that we can use the induction hypothesis on $\psi(\overline{x}, \overline{z})$, because all variables in ψ must effectively occur in α and therefore $|\overline{x}| + |\overline{z}| \leqslant m$. Now, let θ_1 be $\bigwedge_{i=1}^{r} x_i = y_i$ and let θ_2 be $\bigwedge_{j=1}^{r} x_{i_j} = y_j$. Then, $E_\varphi^{(x_1,\ldots,x_k)}$ is

$$G_k \ltimes_{\theta_2} (E_\alpha^{(x_{i_1},\ldots,x_{i_r},z_1,\ldots,z_p)} \ltimes_{\theta_1} E_\psi^{(x_{i_1},\ldots,x_{i_r},z_1,\ldots,z_p)}).$$

□

3.3 Decidability and complexity

Note that our translation from SA to GF is effectively computable. Thus, any decidability result of GF carries over to SA. In particular, by the decidability of GF [4, 23], we obtain:

Corollary 3.6. *Satisfiability of SA expressions is decidable.*

By the finite model property of GF [3], we obtain:

Corollary 3.7. *The semijoin algebra has the finite model property.*

For a fixed finite vocabulary—in logic, the names and arities of relation names are collected in a "vocabulary" instead of in a database schema—the satisfiability problem for GF is in EXPTIME [23]. Note that our translation from SA to GF is exponential in general, so an EXPTIME complexity result for SA does not directly follow from Theorem 3.4. Nevertheless, we have the following:

Theorem 3.8. *For every fixed database schema* $\mathbf{S}$*, the satisfiability problem for SA is in* EXPTIME.

Proof. Given an SA expression E of arity k over $\mathbf{S}$, we apply the same translation procedure as in Theorem 3.4, but we use a new k-ary relation symbol H_k instead of formula $\mathbb{G}_k$. From the translation it is clear that if $\mathbb{G}_k$ is used, then $k \leqslant m$, where m is the maximal arity of relation symbols in $\mathbf{S}$. The translation thus gives us a GF formula $\varphi'_E(x_1, \ldots, x_k)$ over $\mathbf{S}' := \mathbf{S} \cup \{H_1, \ldots, H_m\}$. Now consider the following sentence over $\mathbf{S}'$:

$$\zeta := \bigwedge_{k=1}^{m} \forall \overline{x}(\mathbb{G}_k(\overline{x}) \to H_k(\overline{x})) \ \wedge \ \bigwedge_{k=1}^{m} \forall \overline{x}(H_k(\overline{x}) \to \mathbb{G}_k(\overline{x}))$$

By Lemma 3.3, ζ is in GF. We now prove the following: E is satisfiable if and only if $\varphi'_E(x_1, \ldots, x_k) \wedge \zeta$ is satisfiable.

Let $\overline{a} \in E(D)$. By Theorem 3.4, $D \models \varphi_E(\overline{a})$. Define D' as the $\mathbf{S}'$-structure with $H_k(D') = \mathbb{G}_k(D)$, for all k. On all $\mathbf{S}$-relations R, $R(D)$ and $R(D')$ coincide. It is now clear that $D' \models \varphi'_E(\overline{a}) \wedge \zeta$.

For the other direction, let $D' \models \varphi'_E(\overline{a}) \wedge \zeta$. From the definition of ζ, it follows that $H_k(D') = \mathbb{G}_k(D')$, for all k. Therefore, $D \models \varphi_E(\overline{a})$ and thus, by Theorem 3.4, $\overline{a} \in E(D)$.

Note that ζ depends only on $\mathbf{S}$ and is thus constant, and that φ'_E is computable from E in polynomial time. We have thus reduced the satisfiability problem for SA in polynomial time to the satisfiability problem for GF. □

Now consider the translation from GF to SA in the proof of Theorem 3.5. As this translation is linear, we can transfer lower complexity bounds known for GF. But some care has to be taken because we consider database schemas, which are *finite* vocabularies, and Grädel's proof of EXPTIME-hardness for GF [23] considers an infinite (though bounded-arity) vocabulary. In fact, it is not hard to see that satisfiability for the guarded fragment with only unary predicates is NP-complete.

Theorem 3.9. *For every fixed database schema* $\mathbf{S}$ *with at least one relation symbol of arity two, the satisfiability problem for SA is* EXPTIME*-hard.*

Proof. We give a sketch only. EXPTIME-hardness of the guarded fragment can be shown by an encoding of the local-global satisfiability problem for modal logic **K**. (Given two formulas ϕ and ψ, is ϕ satisfiable in a Kripke model in which ψ holds in every world? [50].) Every modal formula is locally equivalent to the guarded formula obtained by the standard translation. Whence we obtain EXPTIME-hardness for vocabularies with an unbounded number of unary predicates and one binary predicate. Using a technique described by Halpern [35] we can reduce the number of propositional variables to just one and obtain an equisatisfiable formula. In the equivalent guarded formula we can now replace the unary predicate Px by $R(x, x)$, and again obtain an equisatisfiable formula. Whence the result. □

Combining Theorems 3.8 and 3.9, we obtain

Theorem 3.10. *For every fixed database schema* $\mathbf{S}$ *with at least one relation symbol of arity two, the satisfiability problem for SA is* EXPTIME*-complete.*

3.4 Fixed point extensions

In this section we define the fixed point extension μSA of SA and show that it corresponds to μGF in the same way that SA corresponds to GF. We recall the definition of guarded fixed point logic μGF [26]. For background on fixed point logics, we refer to Ebbinghaus and Flum [17].

Definition 3.11 (μGF)**.** The guarded fixed point logic μGF is obtained by adding to GF the following rules for constructing fixed-point formulae:

Let W be a k-ary relation variable and let $\overline{x} = (x_1, \ldots, x_k)$ be a k-tuple of distinct variables. Further, let $\psi(W, \overline{x})$ be a guarded formula where W appears only positively and not in guards. Moreover we require that all the free variables of $\psi(W, \overline{x})$ are contained in $\overline{x}$. For such a formula $\psi(W, \overline{x})$ we can build the formula $[\mathrm{LFP}\ W\overline{x}.\psi](\overline{x})$.

The semantics of the fixed point formulae is the usual one: Given a database D and a valuation χ for the free second-order variables in ψ, other than W, the formula $\psi(W, \overline{x})$ defines an operator on k-ary relations $W \subseteq \mathbb{U}^k$, namely $\psi^{D,\chi} := \{\overline{a} \in \mathbb{U}^k \mid D, \chi \models \psi(W, \overline{a})\}$. Since W occurs only positively in ψ, this operator is monotone and therefore has a least fixed point $\mathrm{LFP}(\psi^{D,\chi})$. Now, the semantics of a least fixed point formula is defined by $D, \chi \models [\mathrm{LFP}\ W\overline{x}.\psi(W, \overline{x})](\overline{a})$ iff $\overline{a} \in \mathrm{LFP}(\psi^{D,\chi})$.

Correspondingly, we will now define the fixed point extension μSA of SA. We assume a database schema V disjoint from $\mathbf{S}$. The relation names in V will be called

relation variables. For each μSA expression E, we also define the set $F(E)$ of free relation variables in E and the sets $\mathrm{pos}(E)$ and $\mathrm{neg}(E)$ that contain the relation variables that occur positively and negatively in E, respectively.

Definition 3.12 (μSA)**.** The syntax and semantics of μSA are inductively defined as follows:

1. Each relation symbol $R \in \mathbf{S}$ is in μSA. $F(R) = \varnothing$, $\mathrm{pos}(R) = \{R\}$ and $\mathrm{neg}(R) = \varnothing$. Its arity comes from $\mathbf{S}$.

2. Each relation variable $X \in V$ is in μSA. $F(X) = \{X\}$, $\mathrm{pos}(X) = \{X\}$ and $\mathrm{neg}(R) = \varnothing$. Its arity comes from V.

3. If $E_1, E_2 \in \mu$SA have arity n, then also $E := E_1 \cup E_2$ belongs to μSA and is of arity n. $F(E) = F(E_1) \cup F(E_2)$, $\mathrm{pos}(E) = \mathrm{pos}(E_1) \cup \mathrm{pos}(E_2)$ and $\mathrm{neg}(E) = \mathrm{neg}(E_1) \cup \mathrm{neg}(E_2)$.

4. If $E_1, E_2 \in \mu$SA have arity n, then also $E := E_1 - E_2$ belongs to μSA and is of arity n. $F(E) = F(E_1) \cup F(E_2)$, $\mathrm{pos}(E) = \mathrm{pos}(E_1) \cup \mathrm{neg}(E_2)$ and $\mathrm{neg}(E) = \mathrm{neg}(E_1) \cup \mathrm{pos}(E_2)$.

5. If $E \in \mu$SA has arity n; if $i, j \in \{1, \ldots, n\}$, and $i_1, \ldots, i_k$ are elements of $\{1, \ldots, n\}$, then $E' := \sigma_{i=j}(E)$ and $E'' := \pi_{i_1,\ldots,i_k}(E)$ belong μSA and are of arity n and k respectively. $F(E') = F(E'') = F(E)$, $\mathrm{pos}(E') = \mathrm{pos}(E'') = \mathrm{pos}(E)$ and $\mathrm{neg}(E') = \mathrm{neg}(E'') = \mathrm{neg}(E)$.

6. If $E_1, E_2 \in \mu$SA have arities n and m, and $\theta(x_1, \ldots, x_n, y_1, \ldots, y_m)$ is a conjunction of equalities of the form $\bigwedge_{l=1}^{s} x_{i_l} = y_{j_l}$, then also $E := E_1 \ltimes_\theta E_2$ belongs to μSA and is of arity n. $F(E) = F(E_1) \cup F(E_2)$, $\mathrm{pos}(E) = \mathrm{pos}(E_1) \cup \mathrm{pos}(E_2)$ and $\mathrm{neg}(E) = \mathrm{neg}(E_1) \cup \mathrm{neg}(E_2)$.

7. If E is a μSA expression such that $X \notin \mathrm{neg}(E)$ and $X \in F(E)$, then also $E' := [\mathrm{LFP}\ X.E]$ is a μSA expression. $F(E') = F(E) - \{X\}$, $\mathrm{pos}(E') = \mathrm{pos}(E) - \{X\}$ and $\mathrm{neg}(E') = \mathrm{neg}(E)$.

Let E be a μSA expression and let D be a database over $\mathbf{S} \cup F(E)$. Then the result of E on D, denoted $E(D)$, is defined inductively as follows:

1. For R in $\mathbf{S} \cup F(E)$, $R(D) := D(R)$.

2. $E_1 \cup E_2(D) := E_1(D) \cup E_2(D)$, $E_1 - E_2(D) := E_1(D) - E_2(D)$.

3. $\sigma_{i=j}E(D) := \{\overline{a} \in E(D) \mid a_i = a_j\}$.

4. $\pi_{i_1,\ldots,i_k}E(D) := \{(a_{i_1}, \ldots, a_{i_k}) \mid (a_1, \ldots, a_n) \in E(D)\}$.

5. $E_1 \ltimes_\theta E_2(D) := \{\overline{a} \in E_1(D) \mid \exists \overline{b} \in E_2(D) : \theta(\overline{a}, \overline{b})\}$.

Let D be a database over $\mathbf{S} \cup F(E) - \{X\}$. Let k be the arity of X. Then $[\mathrm{LFP}\ X.E](D)$ is defined as the least fixed point of the operator E^D on k-ary relations on $\mathbb{U}$, defined as

follows: $E^D(r) := E(D, r)$. Here, by (D, r) we denote the database D' over $\mathbf{S} \cup F(E)$ defined by

$$\begin{cases} D'(R) = D(R) & \text{if } R \in \mathbf{S} \\ D'(Y) = D(Y) & \text{if } Y \in V,\ Y \neq X \\ D'(X) = r \end{cases}$$

This least fixed point always exists because E^D is monotone, as shown in Lemma 3.13.

Lemma 3.13. *Let E be a μSA expression such that $X \in F(E)$. Let D be a database over $\mathbf{S} \cup F(E) - \{X\}$. If $X \notin \mathrm{neg}(E)$, then the operator E^D is monotone; if $X \notin \mathrm{pos}(E)$, then E^D is anti-monotone.*

Proof. The proof is by structural induction on E. Suppose $X \notin \mathrm{neg}(E)$. The base case where $E = X$ is clear. Suppose the lemma is true for E_1 and E_2, then the lemma also holds for $\sigma_{i=j}E_1$, $\pi_{i_1,\ldots,i_k}E_1$, $E_1 \cup E_2$ and $E_1 \ltimes_\theta E_2$ because selection, projection, union and semijoin are monotone operators. If $E = E_1 - E_2$ and $X \notin \mathsf{neg}(E)$, then $X \notin \mathsf{neg}(E_1)$ and $X \notin \mathsf{pos}(E_2)$, so E_1^D is monotone and E_2^D is anti-monotone by induction. Then, clearly E^D is monotone. The case where $X \notin \mathrm{pos}(E)$ is analogous. □

We now prove that μSA and μGF are equivalent in the same way as the logics without fixed point extensions: μSA is subsumed by μGF, and conversely, the result of any μGF query restricted to guarded tuples is always expressible in μSA.

Theorem 3.14. *For every μSA expression E of arity k, there exists a μGF formula $\varphi_E(x_1, \ldots, x_k)$ such that for every database D and for every tuple $\overline{a}$ in $\mathbb{U}^k$, we have $\overline{a} \in E(D)$ iff $D \models \varphi_E(\overline{a})$.*

Proof. The proof is by structural induction. All cases except least fixed point are handled as in the proof of Theorem 3.4. In particular, if $X \in F(E_1)$, then X does not appear in a guard of φ_{E_1}; if $X \notin \mathrm{neg}(E_1)$, then X is positive in φ_{E_1}. Consider now the case where E is of the form [LFP $X.E_1$]. Now, [LFP $X\overline{x}.\varphi_{E_1}(\overline{x})](\overline{x})$ is a well-defined μGF formula equivalent to E. □

To go from μGF to μSA, the following lemma proved by Grädel et al. [25] is particularly instrumental:

Lemma 3.15. *Any formula of μGF is logically equivalent to one in which all fixed points are of the form* [LFP $W\overline{x}.(\psi(\overline{x}) \wedge \mathbb{G}(\overline{x}))](\overline{x})$.

Theorem 3.16. *For every μGF formula $\varphi(x_1, \ldots, x_k)$ with $k \leqslant m$, there exists a μSA expression E_φ such that for every database D and for every guarded tuple $\overline{a}$ in $\mathbb{U}^k$, we have $D \models \varphi(\overline{a})$ iff $\overline{a} \in E_\varphi(D)$.*

Proof. The proof is by structural induction. All cases except least fixed point are handled as in the proof of Theorem 3.5. Consider now the case where $\varphi(x_1, \ldots, x_k)$ is of the form [LFP $W\overline{x}.\psi(\overline{x})](\overline{x})$. By Lemma 3.15, we may assume that $\psi(\overline{x})$ is of the form $\chi(\overline{x}) \wedge \mathbb{G}(\overline{x})$. By induction we have that [LFP $X.E_\chi^{(\overline{x})}$] is equivalent to $\varphi(\overline{x})$. □

Using an argument similar to that of Theorem 3.8, and given that for any fixed database schema, satisfiability for μGF is in EXPTIME [26], we obtain that satisfiability for μSA is in EXPTIME. It is actually EXPTIME-complete, since satisfiability for SA is already EXPTIME-hard (Theorem 3.9).

3.5 Generalizations of GF and SA

The semijoin operator can be seen as a relativized version of the product operator (expressible by a join with an always true join condition, e.g., an empty conjunction); thus, SA is a relativized version of RA. Indeed, let $\mathcal{I}$ be a function mapping pairs (D, k), where D is a database over $\mathbf{S}$ and k is a natural number, to relations, such that $\mathcal{I}(D, k)$ is a k-ary relation on D. Define the syntax and semantics of the relational algebra relativized to $\mathcal{I}$, as follows:

- The syntax is that of the relational algebra;
- The semantics of the selection, projection, union and difference operator are the same as in the relational algebra. The semantics of the product operator relativized to $\mathcal{I}$ is defined as follows:

$$E_1 \times E_2(D) := \{(\overline{a}, \overline{b}) \mid \overline{a} \in E_1(D), \overline{b} \in E_2(D), \\ (\overline{a}, \overline{b}) \in \mathcal{I}(D, \text{arity}(\overline{a}) + \text{arity}(\overline{b}))\}$$

We denote RA relativized to $\mathcal{I}$ by $\text{RA}^{\mathcal{I}}$. Then, if we define $\mathcal{I}^{\text{GF}}(D, k) := G_k(D)$, for all D and k, it is clear that $\text{RA}^{\mathcal{I}^{\text{GF}}}$ is equivalent to SA (and thus also to GF). Indeed, let r and s be relation instances with arities n_r and n_s respectively. Then the relativized product $r \times s$ can be expressed in SA as $(G_{n_r+n_s} \ltimes_{\theta_r} r) \ltimes_{\theta_s} s$, where $\theta_r = \bigwedge_{i=1}^{n_r} x_i = y_i$ and $\theta_s = \bigwedge_{i=1}^{n_s} x_{n_r+i} = y_i$. Furthermore, the semijoin $r \ltimes_\theta s$ with $\theta = \bigwedge_{\ell=1}^{k} x_{i_\ell} = y_{j_\ell}$ can be expressed in $\text{RA}^{\mathcal{I}^{\text{GF}}}$ as $\pi_{1,\ldots,n_r}\sigma_\varphi(r \times \pi_{j_1,\ldots,j_k}s)$, where $\varphi = \bigwedge_{\ell=1}^{k} i_\ell = n_r + j_\ell$.

In literature, generalizations of GF based on loosening the guards have been considered [59, 22, 49]. In the packed fragment for example [49], all quantifications are relative to the set of packed tuples. A tuple $\overline{a}$ is *packed* in a database D over $\mathbf{S}$ if each a_i and a_j appear together in some tuple $\overline{d} \in R(D)$. If we define $\mathcal{I}^{\text{PF}}(D, k) := P_k(D)$, where P_k returns all packed k-tuples in D, then it is easy to adapt our proofs of Theorem 3.4 and 3.5 and show that $\text{RA}^{\mathcal{I}^{\text{PF}}}$ is equivalent to the packed fragment.

3.6 Evaluation complexity

For a fixed database schema $\mathbf{S}$, we can consider the evaluation problem for SA, defined as follows:

Input: A database D over $\mathbf{S}$, a SA expression E and a tuple $\overline{a} \in D$.

Decide: Is $\overline{a} \in E(D)$?

It is known that the corresponding problem for GF is decidable in linear time on a RAM (Random Access Machine), provided a suitable array-based representation is used to represent finite structures [21]. Actually, in that article, this linear evaluation complexity was shown for a language called Datalog LIT, and it is an easy matter to provide a linear translation from SA to Datalog LIT. We can thus conclude:

Theorem 3.17. *For every fixed database schema* $\mathbf{S}$*, the evaluation problem for SA can be solved in linear time.*

A	B
Visits(alex, pareto bar)	Visits(bart, fuel bar)
Serves(pareto bar, westmalle)	Visits(daniel, goof bar)
Likes(alex, westmalle)	Serves(fuel bar, orval)
	Serves(goof bar, westvleteren)
	Likes(bart, westvleteren)
	Likes(daniel, orval)

Figure 3.1: Two databases A and B showing that the query "Is there a drinker that visits a bar that serves a beer he likes?" is not expressible in SA.

3.7 Application

An important application of the Codd theorem for the semijoin algebra is showing that a certain query can not be expressed in SA. To show that a query is not expressible in GF, there is a tool known as "guarded bisimulation". Indeed, Andréka et al. have shown that GF equals the class of first-order formulas invariant under guarded bisimulation [4].

As a corollary of the SA $\subseteq$ GF part (Theorem 3.4) of the Codd theorem and the invariance of GF under guarded bisimulations (see Proposition 2.9), we have:

Corollary 3.18. *If $A, \overline{a} \sim_g B, \overline{b}$, then for any SA expression E we have:*

$$\overline{a} \in E(A) \quad \Leftrightarrow \quad \overline{b} \in E(B).$$

Consider for example, the query Q from Example 3.1 in the beginning of this chapter. Figure 3.1 shows two databases A and B. In A, Alex visits the Pareto bar, which serves Westmalle, which he likes. But in B no drinker visits a bar that serves a beer he likes. Nevertheless, $(A, \text{alex}) \sim_g (B, \text{bart})$, so any SA expression that returns alex on A will also return bart on B and therefore cannot have the semantics of Q. To see that $(A, \text{alex}) \sim_g (B, \text{bart})$, we invite the reader to verify that the following set $\mathcal{I}$ is a guarded bisimulation between A and B:

$$\mathcal{I} = \{\text{alex} \mapsto \text{bart}\} \cup \bigcup \Big\{ \{\overline{a} \mapsto \overline{b} \mid \overline{a} \in A(R) \text{ and } \overline{b} \in B(R)\} \mid R = \text{Visits}, \text{Serves}, \text{Likes} \Big\}$$

3.8 Discussion

Our characterization of the guarded fragment by using semijoins suggests generalizations of GF in directions other than those considered up to now, based on loosening the guards. Specifically, we can allow other semijoin conditions than just conjunctions of equalities.

But, as the following example shows, such generalizations are not innocent. For instance, let us allow nonequalities in semijoin conditions. This variant of the semijoin algebra, denoted SA[qff(=), qff(=)] or also $\mathrm{SA}^{\neq}$, is strictly more expressive than GF. Consider for example the query that asks whether there are at least two distinct elements in a single unary relation S. This is expressible in $\mathrm{SA}^{\neq}$ as $S \ltimes_{x_1 \neq y_1} S$, but is not expressible in GF. Indeed, a set with a single element is "guarded bisimilar" to a set with two elements [4].

Unfortunately, it follows from a result by Grädel that these nonequalities in semijoin conditions make SA undecidable.

Theorem 3.19. *Satisfiability of $SA^{\neq}$ expressions is undecidable.*

Proof. Grädel [23, Theorem 5.8] shows that GF with functionality statements in the form of functional[D], saying that the binary relation D is the graph of a partial function, is a conservative reduction class. Since functional[D] is expressible in $\mathrm{SA}^{\neq}$ as $D \ltimes_{x_1=y_1 \wedge x_2 \neq y_2} D = \varnothing$, it follows that $\mathrm{SA}^{\neq}$ is undecidable. □

A generalization of guarded bisimilarity to the semijoin algebra with arbitrary semijoin conditions is proposed in Chapter 7.

We note that it has already been observed that boolean acyclic non-recursive stratified Datalog (NRSD) programs have the same expressive power as GF sentences [18, 21]. Each rule in such a program is an acyclic join query. By the well-known correspondence between acyclic join queries and semijoin programs [11], these acyclic NRSD programs also correspond to SA. Hence, the correspondence we have shown between SA and GF could also have been derived by combining these previous results. Nevertheless, the equivalence proof we give is direct and elementary.

4

Linear space query processing

In this chapter, we show a dichotomy theorem stating that every relational algebra expression is either linear or quadratic. Furthermore, we will characterize the class of linear relational algebra queries as the class of semijoin algebra queries.

4.1 Introduction

Consider a relational algebra expression to be linear if on every database, the size of every intermediate result is linear in the size of the input database. Examples of linear RA expressions are $\sigma_{1=2}R$, $\pi_{2,3}R - S$, and $R \cup S$. An example of a non-linear RA expression is $R \cup (S \bowtie_{2=2} T)$. Indeed, the size of the join of S and T grows quadratically with S and T.

It is also clear that all operators of RA except for the join produce results of linear size. Therefore, one might think that the linear RA expressions are exactly those that do not use joins, but this is false: Linear expressions exist that do use joins. An example of such an expression is $R \bowtie_{2=1} \pi_1 S = \pi_{1,2,2}(R \ltimes_{2=1} S)$, where R and S are binary relation names. On the other hand, every SA expression is linear. Therefore, the question arises whether there exist other linear queries than SA queries.

In this chapter, we answer this question negatively: we show that every query that can be expressed by a linear RA expression can already be expressed by an SA expression. We will use guarded bisimulations (Chapter 2, Chapter 3) to prove our result. The result says that guarded bisimulations can be used as a tool to show that certain queries can only be expressed by quadratic RA expressions and are therefore hard on the query processor. We will apply the result to division and set (containment and equality) joins: any RA expression for these operators must produce intermediate results of quadratic size. This provides an a posterior justification of work done by various researchers on implementing division and set joins as special-purpose operators,

or on implementing them by compiling to the more powerful version of the relational algebra that includes grouping, sorting, and aggregation operators [37, 48, 52].

4.2 A dichotomy theorem

Before we can state the theorem we need precise definitions of what we mean by "linear" and "quadratic" expressions. Beware that "linear" is an upper-bound notion, while "quadratic" is a lower-bound notion.

Definition 4.1. The *size* of a relation is defined as its cardinality. The *size* of a database D, denoted by $|D|$, is the sum of the sizes of its relations.

For our definitions of linear and quadratic, we will use the familiar O and Ω notation. For a function $f\colon \mathbb{N} \to \mathbb{N}$, recall that $f = O(n)$ if for some $c > 0$ and some n_0, $f(n) \leqslant cn$ for all $n \geqslant n_0$; and $f = \Omega(n^2)$ if for some $c > 0$, $f(n) \geqslant cn^2$ infinitely often [2].

Definition 4.2. For any RA expression E, define the function

$$c(E)\colon \mathbb{N} \to \mathbb{N}\colon n \mapsto \max\{|E(D)| : |D| = n\}.$$

Then E is called

- *linear* if for each subexpression E' of E, $c(E') = O(n)$;
- *quadratic* if for some subexpression E' of E, $c(E') = \Omega(n^2)$.

We will prove:

Theorem 4.3. *Every RA expression is either linear or quadratic.*

In other words, intermediate complexities such as $O(n \log n)$ are not achievable in RA. Anyone who has played long enough with RA expressions will intuitively know that, but we have never seen a proof. Moreover, we also have the following variant:

Theorem 4.4. *Every RA expression that is not quadratic, is equivalently expressible in SA.*

Note that the equi-semijoin operator can be expressed in RA in a linear way; for example, if R and S have arity two, then

$$R \underset{2=1}{\ltimes} S = \pi_{1,2}(R \underset{2=1}{\bowtie} \pi_1(S)).$$

From the above theorems we therefore obtain:

Corollary 4.5. *A query is expressible by a linear RA expression if and only if it is expressible by an SA expression.*

We will prove Theorem 4.3 and Theorem 4.4 simultaneously. Our crucial lemma is Lemma 4.10. In order to state it, we need two definitions.

Definition 4.6. Let E be an RA expression of the form $E_1 \bowtie_\theta E_2$. We view $\theta \equiv \bigwedge_{s=1}^{k} x_{i_s} = y_{j_s}$ as the set of pairs $\{(i_s, j_s) \mid s = 1, \ldots, k\}$. For $\ell = 1, 2$, the sets $\mathsf{constrained}_\ell(E)$ and their complements $\mathsf{unc}_\ell(E)$ are now defined as follows:

$$\begin{aligned}
&\mathsf{constrained}_1(E) := \{i \mid \exists j : (i, j) \in \theta\}\\
&\mathsf{unc}_1(E) := \{1, \ldots, \mathrm{arity}(E_1)\} - \mathsf{constrained}_1(E)\\
&\mathsf{constrained}_2(E) := \{j \mid \exists i : (i, j) \in \theta\}\\
&\mathsf{unc}_2(E) := \{1, \ldots, \mathrm{arity}(E_2)\} - \mathsf{constrained}_2(E)
\end{aligned}$$

Example 4.7. For the expression $E = R \bowtie_{3=1} S$, where R and S are ternary, we get:

$$\begin{aligned}
&\theta = \{(3, 1)\}\\
&\mathsf{constrained}_1(E) = \{3\} \qquad\qquad \mathsf{unc}_1(E) = \{1, 2\}\\
&\mathsf{constrained}_2(E) = \{1\} \qquad\qquad \mathsf{unc}_2(E) = \{2, 3\}.
\end{aligned}$$

□

Definition 4.8. Let D be a database and let E be an RA expression of the form $E_1 \bowtie_\theta E_2$. For any $\overline{a} \in E_1(D)$, we denote the set of elements occurring in $\overline{a}$ by $\mathsf{set}(\overline{a})$. We now define the set of *free values of* $\overline{a}$ as follows:

$$F_1^E(\overline{a}) := \mathsf{set}(\overline{a}) - \{a_i \mid i \in \mathsf{constrained}_1(E)\}$$

The set $F_2^E(\overline{b})$ of free values of a tuple $\overline{b} \in E_2(D)$ is defined analogously.

Example 4.9. Take again expression E from Example 4.7. Suppose that relation R contains the tuples $\overline{a} = (1, 2, 3)$ and $\overline{b} = (4, 5, 4)$, and that relation S contains the tuples $\overline{c} = (3, 4, 5)$ and $\overline{d} = (3, 3, 3)$. Then:

$$\begin{aligned}
&F_1^E(\overline{a}) = \{1, 2\} \qquad\qquad F_2^E(\overline{c}) = \{4, 5\}\\
&F_1^E(\overline{b}) = \{5\} \qquad\qquad F_2^E(\overline{d}) = \varnothing
\end{aligned}$$

□

We can now state the following crucial lemma:

Lemma 4.10. *Let $E = E_1 \bowtie_\theta E_2$, where E_1 and E_2 are SA-expressions. Assume there exists a database D and a tuple $(\overline{a}, \overline{b}) \in E_1 \bowtie_\theta E_2(D)$ such that $F_1^E(\overline{a}) \neq \varnothing$ and $F_2^E(\overline{b}) \neq \varnothing$. Then there exists a sequence $(D_n)_{n \geqslant 1}$ of databases such that for some constant $c > 0$ and for all n:*

1. *$|D_n| \leqslant cn$, and*
2. *$|E_1 \bowtie_\theta E_2(D_n)| \geqslant n^2$.*

Before we prove this lemma, we define the notion of "tuple space" used in the proof.

Definition 4.11. Let D be a database over database schema $\mathbf{S}$. The tuple space T_D of database D is defined as $\bigcup\{D(R) \mid R \in \mathbf{S}\}$.

From the definition of guarded set, it is clear that for each tuple $\overline{d} \in T_D$, we have that $\mathsf{set}(\overline{d})$ is guarded and conversely, for each guarded set X there is a tuple $\overline{d} \in T_D$ with $\mathsf{set}(\overline{d}) = X$.

Proof. We give a proof by construction.

The desired sequence is constructed as follows. For D_1 we take D. For $k \geqslant 1$, we construct D_{k+1} from D_k as follows:

1. for each $x \in F_1^E(\overline{a})$ and for each $x \in F_2^E(\overline{b})$, we choose a fresh new element $\mathsf{new}^{(k)}(x)$ from $\mathbb{U}$, i.e., $\mathsf{new}^{(k)}(x) \in \mathbb{U} - adom(D_k)$.
2. for each tuple $\overline{t} = (t_1, \ldots, t_n) \in T_D$ satisfying $\mathsf{set}(\overline{t}) \cap F_1^E(\overline{a}) \neq \varnothing$, we construct a tuple $f_1^{(k)}(\overline{t}) = (r_1, \ldots, r_n)$ with
$$r_i = \begin{cases} \mathsf{new}^{(k)}(t_i) & \text{if } t_i \in F_1^E(\overline{a}) \\ t_i & \text{else} \end{cases}$$
We put this tuple in D_{k+1} in precisely the same relations as $\overline{t}$. Note that by construction $\overline{t} \mapsto f_1^{(k)}(\overline{t})$ is a partial isomorphism.
3. for each tuple $\overline{t} = (t_1, \ldots, t_n) \in T_D$ satisfying $\mathsf{set}(\overline{t}) \cap F_2^E(\overline{b}) \neq \varnothing$, we construct a tuple $f_2^{(k)}(\overline{t}) = (r_1, \ldots, r_n)$ with
$$r_i = \begin{cases} \mathsf{new}^{(k)}(t_i) & \text{if } t_i \in F_2^E(\overline{b}) \\ t_i & \text{else} \end{cases}$$
We put this tuple in D_{k+1} in precisely the same relations as $\overline{t}$. Note that by construction $\overline{t} \mapsto f_2^{(k)}(\overline{t})$ is a partial isomorphism.

Note that the active domain of D_{k+1} extends the active domain of D_k; also note that the tuple space $T_{D_{k+1}}$ of D_{k+1} extends tuple space T_{D_k} of D_k:

$$\begin{aligned} adom(D_{k+1}) &= adom(D_k) \cup \{\mathsf{new}^{(k)}(x) \mid x \in F_1^E(\overline{a}) \text{ or } x \in F_2^E(\overline{b})\} \\ T_{D_{k+1}} &= T_{D_k} \cup \{f_1^{(k)}(\overline{t}) \mid \overline{t} \in T_D \text{ and } \mathsf{set}(\overline{t}) \cap F_1^E(\overline{a}) \neq \varnothing\} \\ &\quad \cup \{f_2^{(k)}(\overline{t}) \mid \overline{t} \in T_D \text{ and } \mathsf{set}(\overline{t}) \cap F_2^E(\overline{b}) \neq \varnothing\} \end{aligned}$$

To illustrate this construction, let database D be the one shown in the upper part of Figure 4.1 and let expression E be $(R \ltimes_{1=2} T) \bowtie_{3=1} (S \ltimes_{2=1} T)$. Let $\overline{a}$ be $(1, 2, 3)$ and let $\overline{b}$ be $(3, 4, 5)$. Then, $F_1^E(\overline{a}) = \{1, 2\}$ and $F_2^E(\overline{b}) = \{4, 5\}$. For each $i \in F_1^E(\overline{a}) \cup F_2^E(\overline{b})$, we denote $\mathsf{new}^{(1)}(i)$ by i' and $\mathsf{new}^{(2)}(i)$ by i''. Databases D_2 and D_3 are shown in the lower part of Figure 4.1.

Now take $c := 2|D|$. Because in each step at most $2|D|$ tuples are added, the first requirement for the sequence holds.

We now check the second requirement. First, we show that for each n and k with $1 \leqslant k \leqslant n - 1$

$$D, \overline{a} \ \sim_g \ D_n, f_1^{(k)}(\overline{a})$$

Take an arbitrary n and consider the set $\mathcal{I} = \{g_{\overline{t}}^{(k)} \mid \overline{t} \in T_D \text{ with } \mathsf{set}(\overline{t}) \cap F_1^E(\overline{a}) \neq \varnothing, 1 \leqslant k \leqslant n - 1\} \cup \{h_{\overline{t}} \mid \overline{t} \in T_D\}$, where

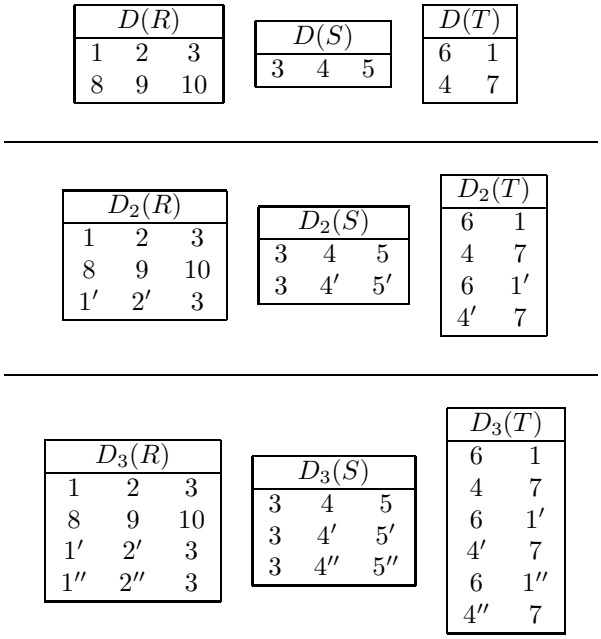


Figure 4.1: Databases $D = D_1$, D_2 and D_3 in the construction for $E = (R \ltimes_{1=2} T) \bowtie_{3=1} (S \ltimes_{2=1} T)$.

- $g_{\overline{t}}^{(k)} : \overline{t} \mapsto f_1^{(k)}(\overline{t})$, and
- $h_{\overline{t}} : \overline{t} \mapsto \overline{t}$.

In our running example, $\mathcal{I} = \{$

$$\begin{array}{lll}
(1,2,3) \mapsto (1',2',3), & (6,1) \mapsto (6,1'), & (1,2,3) \mapsto (1,2,3),\\
(1,2,3) \mapsto (1'',2'',3), & (6,1) \mapsto (6,1''), & (3,4,5) \mapsto (3,4,5),\\
(3,4,5) \mapsto (3,4',5'), & (7,4) \mapsto (7,4'), & (6,1) \mapsto (6,1),\\
(3,4,5) \mapsto (3,4'',5''), & (7,4) \mapsto (7,4''), & (7,4) \mapsto (7,4),\\
 & & (8,9,10) \mapsto (8,9,10)\}.
\end{array}$$

From the construction it follows that each of these functions is a partial isomorphism between D and D_n. Now we check the back and forth properties of $\mathcal{I}$:

Forth. Take an arbitrary partial isomorphism f in $\mathcal{I}$ and an arbitrary guarded set X' in D. Let $\overline{t}'$ be a tuple in T_D such that $\mathsf{set}(\overline{t}') = X'$. Suppose f is $g_{\overline{t}}^{(k)}$ for some $\overline{t}$ and k. We distinguish 2 cases: *i)* $X' \cap F_1^E(\overline{a}) \neq \varnothing$. Then, f agrees with partial isomorphism $g_{\overline{t}'}^{(k)}$ on $\mathsf{set}(\overline{t}) \cap X'$. Indeed, they both map values $x \in F_1^E(\overline{a})$ onto $\mathsf{new}^{(k)}(x)$ and they map values $y \notin F_1^E(\overline{a})$ onto y. *ii)* $X' \cap F_1^E(\overline{a}) = \varnothing$. Then, f agrees with $h_{\overline{t}'}$ on $\mathsf{set}(\overline{t}) \cap X'$. When f is $h_{\overline{t}}$ for some $\overline{t}$, f clearly agrees with $h_{\overline{t}'}$ on $\mathsf{set}(\overline{t}) \cap X'$.

Back. Take an arbitrary partial isomorphism f in $\mathcal{I}$ and an arbitrary guarded set Y' in D_n. We distinguish 2 cases: *i)* $Y' = \mathsf{set}(f_1^{(l)}(\overline{u}))$ for some $1 \leqslant l \leqslant n-1$ and $\overline{u} \in T_D$; and *ii)* $Y' = \mathsf{set}(\overline{t}')$ for some $\overline{t}' \in T_D \cap T_{D_n}$. In case *i)*, f^{-1} agrees with $(g_{\overline{u}}^{(l)})^{-1}$ on $\mathsf{set}(f(\overline{t})) \cap Y'$. In case *ii)*, f^{-1} agrees with $(h_{\overline{t}'})^{-1}$ on $\mathsf{set}(f(\overline{t})) \cap Y'$.

Furthermore, for each $1 \leqslant k \leqslant n-1$, $\overline{a} \mapsto f_1^{(k)}(\overline{a})$ is an element of $\mathcal{I}$. A similar argument leads to

$$D, \overline{b} \ \sim_g \ D_n, f_2^{(k)}(\overline{b})$$

for each $1 \leqslant k \leqslant n-1$.

By Corollary 3.18 we have that for each $0 \leqslant k, l \leqslant n-1$: $f_1^{(k)}(\overline{a}) \in E_1(D_n)$ and $f_2^{(k)}(\overline{b}) \in E_2(D_n)$, where for simplicity we define $f_1^{(0)}$ and $f_2^{(0)}$ as the identity function.

In our running example, only $(1,2,3)$ satisfies $R \ltimes_{1=2} T$ in D, but in D_3 also $(1',2',3)$ and $(1'',2'',3)$ satisfy this expression; also in D_3 the tuples $(3,4,5)$, $(3,4',5')$ and $(3,4'',5'')$ satisfy $S \ltimes_{2=1} T$.

We now show that each pair of tuples $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ with $0 \leqslant k, l \leqslant n-1$ satisfies θ. Let $(i,j) \in \theta$ and let $k, l \in \{0, \ldots, n-1\}$. Then $i \in \mathsf{constrained}_1(E)$ and $j \in \mathsf{constrained}_2(E)$. By construction, the i-th component of $f_1^{(k)}(\overline{a})$ equals a_i and the j-th component of $f_2^{(l)}(\overline{b})$ equals b_j. Because $(\overline{a}, \overline{b})$ satisfies θ, we have $a_i = b_j$. Thus, the i-th component of $f_1^{(k)}(\overline{a})$ equals the j-th component of $f_2^{(l)}(\overline{b})$. We conclude that $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ also satisfies θ. This gives us at least n^2 tuples in $E_1 \bowtie_\theta E_2(D_n)$, which completes the proof. □

Using Lemma 4.10, we can now prove Theorems 4.3 and 4.4. By structural induction, we will prove that any RA expression that is not quadratic, is linear and equivalently expressible in SA.

The base case is clear: R is not quadratic, is linear, and is in SA. For the case of selection, consider an expression of the form σE that is not quadratic (the actual selection condition does not matter here). Then E is not quadratic either, and by induction, E is linear and equivalently expressible in SA as E'. We conclude that σE is linear and equivalently expressible in SA as $\sigma E'$. The cases of projection, union and difference are handled similarly.

The only nonstraightforward case is $E = E_1 \bowtie_\theta E_2$. Assume E is not quadratic. Then, neither E_1 nor E_2 is quadratic, and by induction, E_1 and E_2 are linear and equivalently expressible in SA as E_1' and E_2', respectively. Because E is not quadratic, the conditions of Lemma 4.10 cannot be satisfied. Hence, we know that for each database D and each joining pair of tuples $(\overline{a}, \overline{b})$ in $E_1'(D) \bowtie_\theta E_2'(D)$, either $F_1^E(\overline{a})$ or $F_2^E(\overline{b})$ is empty (or both). If $F_1^E(\overline{a})$ is empty, $\overline{a}$ can be completely retrieved from $E_2'(D)$; if $F_2^E(\overline{b})$ is empty, $\overline{b}$ can be completely retrieved from $E_1'(D)$. E can thus be written as $Z_1 \cup Z_2$, where

$$\begin{aligned} Z_1 &= \{(\overline{a}, \overline{b}) \in E_1' \bowtie_\theta E_2' \mid F_1^E(\overline{a}) = \varnothing\} \\ Z_2 &= \{(\overline{a}, \overline{b}) \in E_1' \bowtie_\theta E_2' \mid F_2^E(\overline{b}) = \varnothing\} \end{aligned} \tag{4.1}$$

We can now express Z_1 and Z_2 in SA, as follows:

$$Z_2 = \bigcup_{f\colon \mathsf{unc}_2(E) \to \mathsf{constrained}_2(E)} \pi_{\overline{p}}(E_1' \ltimes_\theta \sigma_\varphi E_2')$$

Here,

$$\varphi \equiv \bigwedge_{j \in \mathsf{unc}_2(E)} j = f(j)$$

and $\overline{p} = 1, \ldots, \mathrm{arity}(E_1'), g(1), \ldots, g(\mathrm{arity}(E_2'))$ where

$$g(j) = \begin{cases} \min\{i \mid (i,j) \in \theta\} & \text{if } j \in \mathsf{constrained}_2(E) \\ \min\{i \mid (i, f(j)) \in \theta\} & \text{if } j \in \mathsf{unc}_2(E) \end{cases}$$

The use of the minimum function is arbitrary here; any function that chooses an element out of a set will do.

The SA expression for Z_1 is entirely analogous. Since SA expressions are always linear, it also follows that E is linear, as desired. This concludes the proof of Theorems 4.3 and 4.4.

4.3 Division, set join, and friends

By Corollary 4.5, to prove that a query can only be expressed in the relational algebra by quadratic expressions, it suffices to show that it is not expressible in SA. And to show nonexpressibility in SA, we have guarded bisimilarity (see Corollary 3.18) as a tool.

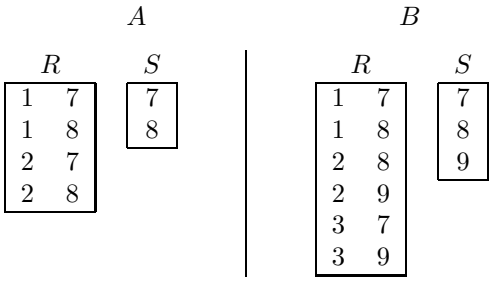


Figure 4.2: Two databases A and B showing that division is inexpressible in SA.

We are thus fully armed now to justify the work done by various authors on implementing division and set join directly as special-purpose operators, or on implementing them by compiling to the more powerful version of the relational algebra that includes grouping, sorting, and aggregation operators [37, 48, 52]:

Proposition 4.12. *Division is expressible in RA only by quadratic expressions. Furthermore, every RA expression that is empty if and only if the set join is empty, must be quadratic.*

Note that it would not be very interesting to claim that the set join itself can only be expressed by quadratic expressions, because the output size of the set join is already quadratic.

To prove Proposition 4.12, we need to show that $R \div S$ is not expressible in SA. Thereto, consider the databases A and B shown in Figure 4.2. Then $R \div S$ equals $\{1, 2\}$ in A, but is empty in B (regardless of whether we use the set containment, or the set equality variant of division). Nevertheless, $A, 1 \sim_g B, 1$, so any SA expression that returns 1 on A will also return 1 on B and therefore cannot express $R \div S$. To see that $A, 1 \sim_g B, 1$, we invite the reader to verify that the following set $\mathcal{I}$ is a guarded bisimulation:

$$\mathcal{I} = \{1 \mapsto 1\} \cup \{\overline{a} \mapsto \overline{b} \mid \overline{a} \in A(R) \text{ and } \overline{b} \in B(R), \text{ or } \overline{a} \in A(S) \text{ and } \overline{b} \in B(S)\}$$

For the set-join version of Proposition 4.12, consider the databases A and B as in Figure 4.2 where now a column with always the same value 4 is inserted into relation S (this will be the first column of the new relation). Then the above $\mathcal{I}$ is still a guarded bisimulation.

Set joins with other set predicates As mentioned in the Introduction in Chapter 1, for both division and set join, any other predicate on sets could as well be used in the place of $\supseteq$ or $=$ [53, 55]. For example, if $R(A, B)$ and $S(C, D)$ are relations and P is a binary set predicate, then we can define the set join with predicate P between R and S as follows:

$$R \bowtie_P^{\text{set}} S := \big\{(a, c) \mid P\big(\{b \mid R(a, b)\}, \{d \mid S(c, d)\}\big)\big\}.$$

Thus, set-containment and set-equality join are set joins with predicate $P(X, Y) := X \supseteq Y$, and $P(X, Y) := X = Y$, respectively. The set join with predicate $P(X, Y) :=$

$X \cap Y \neq \varnothing$ is the standard equijoin. While the set-containment and set-equality join are not expressible by a linear RA expression, the (emptiness test for the) standard equijoin *is* expressible by a linear RA expression. It is therefore an interesting open question for which predicates P the set join is linear.

Other queries Clearly, the applicability of the techniques we have developed in this chapter is not restricted to division and set joins! For example, over the beer-drinkers database schema, consider the query Q from Example 3.1:

Is there a drinker that visits a bar that serves a beer he likes?

We showed in Section 3.7 that Q can not be expressed in SA. Therefore, any RA expression of this query must be quadratic.

4.4 Generalizing the dichotomy

In this section, we will generalize the dichotomy theorem to relational algebra queries where besides the equality predicate "=" also an order predicate "<" is available in selection and join conditions. So, we assume that there is a linear order $<$ on the elements in the universe $\mathbb{U}$.

The part of finite model theory that studies the expressive power of logics over finite structures that are embedded into infinite ones—here, a finite database is embedded into the structure with universe $\mathbb{U}$ equipped with a linear order—is called "embedded finite model theory". Embedded finite model theory has been studied extensively [46].

It is well known that relational algebra expressions that can use a given linear order can express more order-invariant queries than relational algebra expressions that do not use the linear order [1, Exercise 17.27]. Therefore, an interesting question is whether Theorem 4.4 still holds in this more powerful setting with order. We will prove:

Theorem 4.13. *Every* $\mathrm{RA}^{<,<}$ *expression that is not quadratic, is equivalently expressible in* $\mathrm{SA}^{<,=}$.

The proof of this dichotomy theorem is similar to the proof of Theorem 4.4. For example, the non-quadratic $\mathrm{RA}^{<,<}$ expression $\sigma_{i<j}E$ can be expressed in $\mathrm{SA}^{<,=}$ as $\sigma_{i<j}E'$, where E' is the $\mathrm{SA}^{<,=}$ expression equivalent to the non-quadratic expression E, which exists by induction. The non-trivial case again is the non-quadratic join $E = E_1 \bowtie_\theta E_2$, where E_1 and E_2 are $\mathrm{SA}^{<,=}$ expressions. We write E as $\bigcup_{\xi\in\Xi} E_1 \bowtie_\xi E_2$, where $\bigvee_{\xi\in\Xi}\xi$ is θ written in disjunctive normal form (i.e., each ξ is a conjunction of atomic and negated atomic formulas over $\{=,<\}$). It is clear that if E is not quadratic, then $E_\xi := E_1 \bowtie_\xi E_2$ is not quadratic, for all $\xi \in \Xi$. Note that the conditions in Ξ have the form

$$\xi = \bigwedge_{s=1}^{k} i_s \ \alpha_s \ j_s \text{ with } \alpha_s \in \{=, \neq, <, \not<\}, \tag{4.2}$$

with $i_s \in \{1, \ldots, \mathrm{arity}(E_1)\}$, and $j_s \in \{1, \ldots, \mathrm{arity}(E_2)\}$.

We now prove that Lemma 4.10 is still valid for $\mathrm{SA}^{<,=}$ expressions E_1 and E_2 and for conditions ξ of the above form. First, we need to define for such expressions E_ξ the sets $\mathsf{constrained}_\ell(E_\xi)$ and $\mathsf{unc}_\ell(E_\xi)$ for $\ell = 1, 2$.

Definition 4.14. Let E_ξ be an RA expression of the form $E_1 \bowtie_\xi E_2$ where $E_1, E_2 \in \mathrm{SA}^{<,=}$, and where $\xi = \bigwedge_{s=1}^{k} i_s\ \alpha_s\ j_s$ with $\alpha_s \in \{=, \neq, <, \not<\}$. For $\alpha \in \{=, \neq, <, \not<\}$, we define ξ^α as the set of pairs $\{(i_s, j_s) \mid \alpha_s \text{ is } \alpha,\ s = 1, \ldots, k\}$. For $\ell = 1, 2$, the sets $\mathsf{constrained}_\ell(E_\xi)$ and their complements $\mathsf{unc}_\ell(E_\xi)$ are now defined as follows:

$$\mathsf{constrained}_1(E_\xi) := \{i \mid \exists j : (i, j) \in \xi^=\}$$
$$\mathsf{unc}_1(E_\xi) := \{1, \ldots, \mathrm{arity}(E_1)\} - \mathsf{constrained}_1(E_\xi)$$
$$\mathsf{constrained}_2(E_\xi) := \{j \mid \exists i : (i, j) \in \xi^=\}$$
$$\mathsf{unc}_2(E_\xi) := \{1, \ldots, \mathrm{arity}(E_2)\} - \mathsf{constrained}_2(E_\xi)$$

For an RA expression E_ξ, a database D, and tuples $\overline{a} \in E_1(D)$ and $\overline{b} \in E_2(D)$, the sets of free values $F_1^{E_\xi}(\overline{a})$ and $F_2^{E_\xi}(\overline{b})$ are defined identically as for ordinary RA expressions (Definition 4.8).

To show Lemma 4.10 in this new setting, we also need to adapt the notion of guarded bisimulation to accommodate for the order $<$. A $<$-guarded bisimulation is a non-empty set $\mathcal{I}$ of finite $<$-partial isomorphisms satisfying the same back and forth properties as ordinary guarded bisimulations (Definition 2.7). We use the notation $A, \overline{a} \ \sim_g^< \ B, \overline{b}$ to denote that $A, \overline{a}$ and $B, \overline{b}$ are $<$-guarded bisimilar.

Definition 4.15 ($<$-partial isomorphism)**.** Let A and B be two databases over schema $\mathbf{S}$. For $X, Y \subseteq \mathbb{U}$, a mapping $f\colon X \to Y$ is a *$<$-partial isomorphism* from A to B if it is a partial isomorphism from A to B, and moreover, for all x_1 and x_2 in X, we have $x_1 < x_2$ if and only if $f(x_1) < f(x_2)$.

Concerning the proof of Lemma 4.10, now remark the following:

1. We can assume that, in each step k in the construction, each new domain element $\mathsf{new}^{(k)}(x)$ of D_{k+1} and x itself can be chosen to have the same relative order with respect to the other elements in $\mathbb{U}$. This is clear for the case where the order $<$ is dense in $\mathbb{U}$ (e.g., if $\mathbb{U}$ is the set of real numbers). But it is possible *in general* to choose $\mathsf{new}^{(k)}(x)$ with the same relative order as x. If in some step k this were not possible, then create an isomorphic copy D'_k of D_k such that for any two values r, s in D'_k with $r < x < s$, there exists $u \in \mathbb{U}$ different from x such that $r < u < s$.

   To illustrate this, consider the running example from the proof (see Figure 4.1). If the underlying universe is the set of reals, then we could choose $\mathsf{new}^{(1)}(1) = 1' = 1.5$ and $\mathsf{new}^{(2)}(1) = 1'' = 1.75$, and similarly for the other values. If the underlying universe is the set of integers, then we could replace D_1 by the isomorphic database D'_1 obtained by multiplying each value in each tuple by a factor 2.

2. It is easy to see that if $\mathsf{new}^{(k)}(x)$ and x have the same relative order with respect to the other elements in $\mathbb{U}$, then each mapping $\overline{t} \mapsto f_1^{(k)}(\overline{t})$ and each mapping

$\overline{t} \mapsto f_2^{(k)}(\overline{t})$ is a $<$-partial isomorphism from D to D_{k+1}. Therefore, as the back and forth properties of guarded and $<$-guarded bisimulations are identical, we obtain that

$$D, \overline{a} \ \sim_g^< \ D_n, f_1^{(k)}(\overline{a}), \text{ and}$$
$$D, \overline{b} \ \sim_g^< \ D_n, f_2^{(k)}(\overline{b})$$

for each $1 \leqslant k \leqslant n-1$.

3. $\mathrm{SA}^{<,=}$ is invariant under $<$-guarded bisimulation: If $A, \overline{a} \ \sim_g^< \ B, \overline{b}$, then for any $\mathrm{SA}^{<,=}$ expression E we have: $\overline{a} \in E(A) \Leftrightarrow \overline{b} \in E(B)$. The proof is by structural induction.

4. We thus obtain n tuples $f_1^{(k)}(\overline{a})$ (for $k = 0, \ldots, n-1$) in $E_1(D_n)$ and n tuples $f_2^{(l)}(\overline{b})$ (for $l = 0, \ldots, n-1$) in $E_2(D_n)$. We now show that each pair of tuples $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ with $0 \leqslant k, l \leqslant n-1$ satisfies ξ.

   Let $(i,j) \in \xi^=$ and let $k, l \in \{0, \ldots, n-1\}$. Then, an identical argument as in the proof of Lemma 4.10 leads to the pair $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ satisfying $\xi^=$.

   The pair of tuples $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ also satisfies $\xi^<$. Let $(i,j) \in \xi^<$. By construction, the i-th component of $f_1^{(k)}(\overline{a})$ equals either a_i or $\mathsf{new}^{(k)}(a_i)$, and the j-th component of $f_2^{(l)}(\overline{b})$ equals either b_j or $\mathsf{new}^{(l)}(b_j)$. Because $(\overline{a}, \overline{b})$ satisfies ξ, we have $a_i < b_j$. By choosing $\mathsf{new}^{(k)}(a_i)$ and $\mathsf{new}^{(l)}(b_j)$ with the same relative order as a_i and b_j, respectively, we also have $\mathsf{new}^{(k)}(a_i) < b_j$, $a_i < \mathsf{new}^{(l)}(b_j)$, and $\mathsf{new}^{(k)}(a_i) < \mathsf{new}^{(l)}(b_j)$. The arguments that $(f_1^{(k)}(\overline{a}), f_2^{(l)}(\overline{b}))$ satisfies $\xi^{\neq}$ and $\xi^{\not<}$ are similar.

We now return to the non-quadratic join $E = E_1 \bowtie_\theta E_2$, written as $\bigcup_{\xi \in \Xi} E_1 \bowtie_\xi E_2 = \bigcup_{\xi \in \Xi} E_\xi$. Each E_ξ can be written in $\mathrm{SA}^{<,=}$ as $Z_1 \cup Z_2$, where Z_1 and Z_2 are as in Equation 4.1. Here, Z_2 can be written in $\mathrm{SA}^{<,=}$ as follows:

$$Z_2 = \bigcup_{f\colon \mathsf{unc}_2(E) \to \mathsf{constrained}_2(E)} \pi_{\overline{p}}\big(\sigma_\psi(E_1' \ltimes_{\xi^=} \sigma_\varphi E_2')\big)$$

where f, $\overline{p}$, and φ are as in the proof of Theorem 4.4, and where

$$\psi \equiv \bigwedge_{\alpha \in \{\neq, <, \not<\}} \bigwedge_{(i,j) \in \xi^\alpha} i \ \alpha \ g(j),$$

where g is also defined as in the proof of Theorem 4.4.

This concludes the proof of Theorem 4.13.

Note that the expressive power of the relational algebra with order in join conditions collapses to the expressive power of the relational algebra with order in selection conditions. In fact, any join $R \bowtie_\theta S$ can equivalently be written as $\sigma_\theta(R \bowtie_\varphi S)$, where φ is the formula `true` (expressed as an empty conjunction), so that one does not need any predicates at all in join conditions. But it is still an interesting question whether order in join conditions, and more generally, arbitrary quantifier-free formulas over

$\{=, <\}$, allows one to express more *linear* relational algebra queries. We have answered this question negatively in this section. Indeed, according to Theorem 4.13 every query expressible by a linear RA[qff$(=, <)$, qff$(=, <)$] expression is already expressible by an SA[qff$(=, <)$, cf$(=)$] expression; and an SA[qff$(=, <)$, cf$(=)$] expression can be expressed linearly in RA[qff$(=, <)$, cf$(=)$].

Remark 4.16. If we allow constants in selection and join conditions in both RA and SA, then a generalization of the dichotomy in Theorem 4.4 as in Theorem 4.13 does not hold. Consider for example the RA expression $E = R \bowtie_{2=1} \sigma_{1='b' \wedge 2='c'} S$, where R and S are binary relations. E is clearly linear, but can not be expressed in SA because SA expressions can only return guarded tuples, even when constants can be used in selection and semijoin conditions (cf. Lemma 3.2). The output of E, however, can contain non-guarded tuples. Indeed, let $D(R) = \{(a, b)\}$ and let $D(S) = \{(b, c)\}$. Then $E(D) = \{(a, b, b, c)\}$. The tuple (a, b, b, c) is not guarded in database D. Therefore, E can not be expressed in SA, not even when constants can be used in selection conditions.

We should note that a dichotomy in the style of Theorem 4.4 is still valid in the setting where the order predicate and constants can be used in selection and join conditions, but then in the semijoin algebra a constant-tagging operator is needed. We have published this particular dichotomy in a journal article [43]. □

Division

From Theorem 4.13 it follows that division can also not be expressed by a linear $\mathrm{RA}^{<,<}$ expression. Indeed, databases A and B shown in Figure 4.2 are also $<$-guarded-bisimilar. (Here, we take the natural numbers as our universe $\mathbb{U}$, with the natural order $<$.) Therefore, A and B can not be distinguished by $\mathrm{SA}^{<,=}$ expressions. Hence the result.

4.5 Discussion

On the technical side, our work leaves open the generalisation where the universe of data elements is not merely equipped with a total order, but where arbitrary predicates are present which can be used in join conditions. One cannot expect our Theorem 4.4 to hold in all such cases, as this will depend on the predicates at hand. A related issue is to investigate the impact of integrity constraints on our results.

Practical query processing uses a more powerful relational algebra including grouping, sorting, and aggregation operators. Proving complexity lower bounds in such a rich setting seems very challenging to us. However, containment-division can be expressed by the linear expression

$$\pi_A\big(\gamma_{A,\mathrm{count}(B)}(R \ltimes_{B=C} S) \underset{\mathrm{count}(B)=\mathrm{count}(C)}{\ltimes} \gamma_{\varnothing,\mathrm{count}(C)} S\big)$$

using grouping (γ) and aggregation (counting). Equality-division can be expressed by an analogous linear RA expression with grouping and counting [27, 28].

5

Linear time query processing

In this chapter, we introduce and analyze *finite cursor machines*, an abstract model of database query processing. In particular, we will study query processing of relational algebra and semijoin algebra expressions.

5.1 Introduction

A finite cursor machine (FCM) works on a number of lists of tuples and can operate in a finite number of *modes* using an *internal memory* in which it can store bit strings. An FCM accesses each list through finitely many cursors, each of which can read one tuple of a list at any time. A list of tuples can be produced as output. The model incorporates certain "streaming" or "sequential processing" aspects by imposing two restrictions: First, the cursors can only move on the lists sequentially in one direction. Thus once the last cursor has left a tuple of a list, this tuple can never be accessed again during the computation. Second, the internal memory is limited. We will formally define the model using the *abstract state machine (ASM)* methodology [31].

Our main results are concerned with evaluating *relational algebra queries* in the finite cursor machine model. 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. *Semijoins* can be computed by finite cursor machines when sorted versions of the database relations are provided as input. Consequently, every query in the semijoin 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 [19]. 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: 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, where n is the size of the input relations. This result is quite sharp, as we will indicate.

We note that finite cursor machines can compute queries beyond the semijoin algebra, and even queries beyond the relational algebra. We will discuss this matter at the end of this chapter.

The chapter is structured as follows: After recalling the basic terminology from many-sorted logic in Section 5.2, the notion of finite cursor machines is introduced in Section 5.3. The power of $O(1)$-FCMs and of $o(n)$-FCMs is investigated in Sections 5.4 and 5.5. Some concluding remarks and open questions can be found in Section 5.6.

5.2 Preliminaries from logic

To formally introduce our computation model, we need some basic notions from mathematical logic such as many-sorted vocabularies, structures, terms, and atomic formulas. We give a quick reminder of these notions in this section.

A *many-sorted vocabulary* is a tuple $\Upsilon = (S, F, P, C, \tau)$, where S is a set of *sorts*; F is a set of *function symbols*; P is a set of *predicate symbols*; and C is a set of *constant symbols*. Moreover, τ is a mapping on $F \cup P \cup C$ that assigns a *function signature* to each function symbol; a *predicate signature* to each predicate symbol; and a sort to each constant symbol. Here, a function signature is an expression of the form $s_1, \ldots, s_k \to s$, where s and the s_i's are sorts; a predicate signature is simply a tuple of sorts. To indicate the value of τ on some symbol ℓ we write $\ell\colon \tau(\ell)$.

A *structure* $\mathcal{A}$ over Υ is a mapping on $S \cup F \cup P \cup C$, giving an interpretation to all the symbols of the vocabulary:

- if s is a sort, then $s^{\mathcal{A}}$ is a set, called the *elements of sort* s.
- if $f\colon s_1, \ldots, s_k \to s$ is a function symbol, then $f^{\mathcal{A}}$ is a function of type $s_1^{\mathcal{A}} \times \cdots \times s_k^{\mathcal{A}} \to s^{\mathcal{A}}$.
- if $p\colon (s_1, \ldots, s_k)$ is a predicate symbol, then $p^{\mathcal{A}}$ is a subset of $s_1^{\mathcal{A}} \times \cdots \times s_k^{\mathcal{A}}$.
- if $c\colon s$ is a constant symbol, then $c^{\mathcal{A}} \in s^{\mathcal{A}}$.

Terms are expressions built up as follows. Every constant symbol is a term. If $t_1\colon s_1, \ldots, t_k\colon s_k$ are terms, and $f\colon s_1, \ldots, s_k \to s$ is a function symbol, then $f(t_1, \ldots, t_k)\colon s$ is also a term. Every term $t\colon s$ evaluates in a structure $\mathcal{A}$ to an element $t^{\mathcal{A}}$ of sort s in the obvious manner.

Atomic formulas are expressions of the form $p(t_1, \ldots, t_k)$, with $p\colon (s_1, \ldots, s_k)$ a predicate symbol and $t_i\colon s_i$ terms. In a structure $\mathcal{A}$, this formula evaluates to the truth value of $(t_1^{\mathcal{A}}, \ldots, t_k^{\mathcal{A}}) \in p^{\mathcal{A}}$.

5.3 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 detailed definition of sequential ASMs is given in the Lipari guide [31], 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 $\Upsilon_{\mathbf{S}}$ (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 Chapter 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.

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

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.

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 database (see Chapter 2) 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 D then starts with the following structure $\mathcal{M}$ over the vocabulary $\Upsilon_0 \cup \Upsilon_{\mathbf{S}} \cup \Upsilon_M$: The interpretation of Element is $\mathbb{U}$; the interpretation of Bitstring is the set of all finite bit strings; and the interpretation of Mode is simply given by the set of modes themselves. For technical reasons, we must assume that $\mathbb{U}$ contains an element $\bot$. For each $R \in \mathbf{S}$, the sort Row_R is interpreted

[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".

by the set $D(R) \cup \{\bot_R\}$; the function $attribute^i_R$ is defined by $(x_1, \ldots, x_k) \mapsto x_i$, and $\bot_R \mapsto \bot$; finally, the function $next_R$ maps each row to its successor in the list, and maps the last row to $\bot_R$. The dynamic symbol *mode* initially is interpreted by the constant *init*; every register contains the empty bit string; 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 $\Upsilon_0 \cup \Upsilon_{\mathbf{S}} \cup \Upsilon_M$, 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 $\Upsilon_0 \cup \Upsilon_{\mathbf{S}} \cup \Upsilon_M$, 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 D if it accepts *every* enumeration of 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 D, the output of M on *any* enumeration of D is an enumeration of the relation $Q(D)$. 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, 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 $\bot$ and the output

mode is equal to *init*. We denote the output of a machine M working on a database D by $M(D)$.

Space restrictions: We define the *size* of a database D as the total number of tuples in D. For considering FCMs whose bit string 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 D of size n, the machine only stores bit strings 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 bit strings of constant length could also be simulated by finitely many *modes*).

Example 5.1. Consider a query Q defined on a ternary relation R over the set of natural numbers $\mathbb{N}$ that returns the sum of the first and second attribute of each row with a third attribute at least 100. 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
```

□

5.3.1 Discussion of the model

Storing bit strings instead of data elements: An important question about our model is the strict separation between data elements and bit strings. Indeed, data elements are abstract entities, and our background structure may contain arbitrary functions and predicates, mixing data elements and bit strings, with the important restriction that the output of a function is always a bit string. At first sight, a simpler way to arrive at our model would be without bit strings, 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 bit strings, however, one can simply impose restrictions on the length of the bit strings stored in registers, and that is precisely what we will do. Of course, the unlimited model with bit strings can also be computationally complete. It suffices that the background structure provides a coding of data elements by bit strings.

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 chapter 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 $\leqslant d$ can be simulated by a $(k \times d)$-cursor FCM with look-ahead 0. Thus, throughout the remainder of this chapter 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 5.23).

5.4 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 5.2. *Let $M_1, \ldots, M_r$ be FCMs working on a schema $\mathbf{S}$, let $\mathbf{S}'$ 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 $\mathbf{S}'$. Then there exists an FCM M working on schema $\mathbf{S}$, such that $M(D) = M_0(D')$, for each database D with schema $\mathbf{S}$ and the database D' that consists of the output relations $M_1(D), \ldots, M_r(D)$.*

The proof is obvious: Each row in a relation R_i of database D' is an output row of a machine M_i working on 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 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 [9], 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 5.3. 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 $1 \leqslant i, j \leqslant w$:

if $mode = check_{i,j}$ and $attribute_R^2(c_R^i) = attribute_S^1(c_S^j)$ then
  par
    $outputmode := out$
    $out_1 := attribute_R^1(c_R^i)$
    $out_2 := attribute_R^2(c_R^i)$
    $out_3 := attribute_S^1(c_S^j)$
    $out_4 := attribute_S^2(c_S^j)$

```
    mode := next-mode_{i,j}
  endpar
endif
```

Here, *next-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-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 can not 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 5.24). Note that the following result is valid for *arbitrary* (but fixed) background structures.

Theorem 5.4. *There is no $O(1)$-FCM that checks for two sets R and S whether $R \cap S \neq \varnothing$.*

Proof. Let M be an $O(1)$-FCM that is supposed to check whether $R \cap S \neq \varnothing$. Without loss of generality, we assume that $\mathbb{U}$ is totally ordered by a predicate $<$ in Υ_0. Using Ramsey's theorem, we can find an infinite set $V \subseteq \mathbb{U}$ 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 [46, 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, obtaining set S', and consider the run of M on R and S'. When there is a cursor on a'_j, there will be no cursor on a_j, and vice versa. Furthermore, the element a'_j has the same relative order as a_j with respect to the other elements in the lists, and therefore any tuple of elements will satisfy the same predicates as the tuple obtained by replacing a_j by a'_j. 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, will 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 5.5. Recall the definition of a join condition $\theta(x_1, \ldots, x_n, y_1, \ldots, y_m)$ from Chapter 2, and assume that the vocabulary Ω includes a total order $<$ on $\mathbb{U}$. 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 a special case, ψ can simply be true.)

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

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 5.6. Let R and S be ternary relations and consider the semijoin $R \ltimes_\theta S$, where θ is the A-allowed join condition $x_1 = y_1 \wedge x_2 > y_2$. 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 5.7. Consider again the semijoin $R \ltimes_\theta S$, where now 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 $s'_2 \geqslant r_2$ 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 $s'_2 \geqslant r''_2$. 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. 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 [19]. Formally, assume that $\mathbb{U}$ 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 D, we mean an FCM that gets all possible sorted orders of all relations of D as input lists. We then summarize the above discussion as follows:

Proposition 5.8. *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.*

Corollary 5.9. *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 5.9.

Example 5.10. Consider the query $Q := \big(\sigma_{x_1<y_2}(R \cup S)\big) \ltimes_{x_2=y_2} \sigma_{x_2<y_1} T$. The expression tree of Q is shown in Figure 5.1 on the left. The query plan obtained by using the construction in the proof of Corollary 5.9 is shown on the right of Figure 5.1. Here, M_α is used to denote the $O(1)$-FCM computing the operator α. □

The simple proof of Corollary 5.9 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 [56].

Example 5.11. 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 5.2), 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

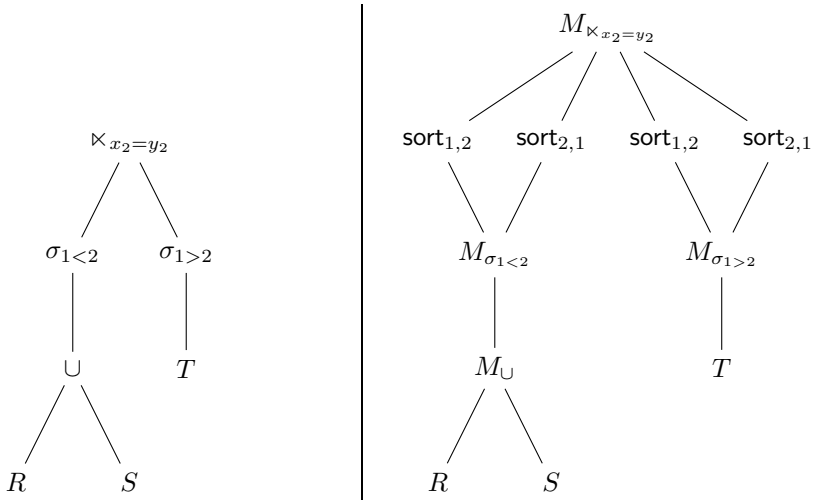


Figure 5.1: On the left: expression tree for SA query $Q = \big(\sigma_{x_1<y_2}(R \cup S)\big) \ltimes_{x_2=y_2} \sigma_{x_1>y_2}T$. On the right: query plan computing Q, composed of $O(1)$-FCMs and sorting operations.

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 $E \underset{\substack{x_2=y_1\\x_1\neq y_2}}{\ltimes} E$ is an abbreviation for the union of the allowed semijoins $E \underset{\substack{x_2=y_1\\x_1<y_2}}{\ltimes} E$ and $E \underset{\substack{x_2=y_1\\x_1>y_2}}{\ltimes} E$.

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 [54].

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

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

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

We recall the following result:

Lemma 5.12 (Hromkovič [40])**.** *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 5.12. To make this text 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 can not read the symbols on the tape.

Lemma 5.13. *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 $2^n/p(n) \geqslant 2$.

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 $y \in L_n^f - P_n^f$:

$$w_1 b w_2 b \cdots b w_i b \cdots b w_n b w_n^R b \cdots b w_i'^R b \cdots b w_2^R b w_1^R$$

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

Proof. Note that as a corollary of Lemma 5.13, 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 $V \subseteq \mathbb{U}$ 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 5.4). 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 $\{(a_i, a_{n-i+1}) \mid 1 \leqslant i \leqslant n\}$ 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 can not 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.

5.5 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 5.14 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 $f\colon \{1,\ldots,p\} \to \{/, \backslash\}$ 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 5.14. 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). In the following proposition, the lower bound was already shown in Theorem 5.14, and the upper bound is easy.

Proposition 5.15. *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 5.5):

Proposition 5.16. *Every semijoin operation with AD-allowed join condition can be computed by an $O(1)$-FCM on AD-sorted inputs.*

Rather than proving this proposition formally, we illustrate it by giving an example.

Example 5.17. 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 $s'_1 \leqslant r_1$. 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 $r'_2 < v$. And so on. □

Remark 5.18. We should note that our notion of allowed join condition (Definition 5.5) 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. □

5.5.1 Intermediate sorting can not 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 extracted from a proof by Nicole Schweikardt and Martin Grohe [29, Theorem 10]. 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 $\{1, \ldots, n\}$:

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

for $1 \leqslant i \leqslant v$ and $1 \leqslant s \leqslant \frac{n}{v}$. 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

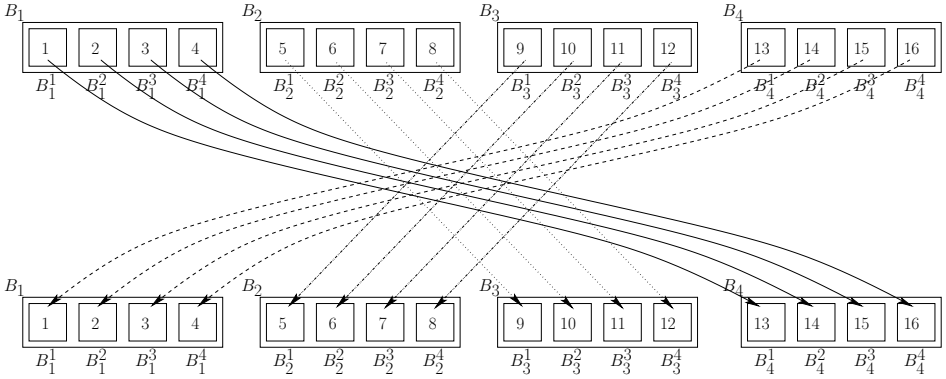


Figure 5.2: Permutation $\pi_{16,4}$.

order but it does not reverse the blocks B_i^j inside B_i in order. The permutation $\pi_{16,4}$ is shown in Figure 5.2.

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 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^1_ℓ, for some $\ell \in B^j_i$) and the other cursor in the pair is on a position in B^{v-j+1}_i on $\overleftarrow{L_2}$ (i.e., the cursor sees a tuple $t^2_{\pi\ell}$, for some $\ell \in B^j_i$), or

- one cursor in the pair is on a position in B^{v-j+1}_{v-i+1} on $\overleftarrow{L_1}$ (i.e., the cursor sees a tuple t^1_ℓ, for some $\ell \in B^j_i$) and the other cursor in the pair is on a position in B^j_{v-i+1} on L_2 (i.e., the cursor sees a tuple $t^2_{\pi\ell}$, for some $\ell \in B^j_i$).

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 $B^{j_0}_{i_0}$ 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 $B^{j_0}_{i_0}$ 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^j_i if at least one pair of cursors of M checks subblock B^j_i.

The above argument thus leads to the following

Lemma 5.19 (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 $B^{j_0}_{i_0}$ 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 5.20 (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 $\mathbf{S}$ be a database schema and let L_1 and L_2 be two distinguished relation names in $\mathbf{S}$. A collection $\mathcal{L}$ of list databases with schema $\mathbf{S}$ 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 list instances of L_1 and L_2 in list database $\mathbf{L}_n(I)$ have length n; and
- the i-th element of the list instance of L_1 in list database $\mathbf{L}_n(I)$ is

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

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

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

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

Lemma 5.21 (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 database containing:*

- *the list instance of L_1 in $\mathbf{L}_n(I)$,*
- *the list instance of L_2 in $\mathbf{L}_n(J)$,*
- *the list instances other than those of L_1 and L_2 in $\mathbf{L}_n(I)$, 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 list databases $\mathbf{L}_n(I)$ and $\mathbf{L}_n(J)$ with $I \neq J$ such that the runs of M on the list instances in both list databases and their reversals are very "similar". In the fooling argument, it is shown that the run of M on the list database $\mathbf{L}_{\text{err}}$ ends in 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 a list database $\mathbf{L}$ — which we will also call *instance* — 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 list databases $\{\mathbf{L}_n(I) \mid I \subseteq \{1, \ldots, n\}\}$. According to the block-checking Lemma 5.19, where block-checking is defined in terms of the lists L_1 and L_2, on each list database $\mathbf{L}_n(I)$ 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 $\frac{2^n/v^2}{2^{n-\frac{n}{v^2}}} = 2^{\frac{n}{v^2}}/v^2$, 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, $2^{\frac{n}{v^2}}/v^2$ 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 bit string 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 $\frac{2^{\frac{n}{v^2}}/v^2}{2^{k \log m + k^2 \log \alpha n + k \cdot r \cdot o(n)}} = 2^{\frac{n}{v^2} - 2\log v - k \log m - k^2 \log \alpha n - k \cdot r \cdot o(n)}$, 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 list database $\mathbf{L}_{\text{err}}$ containing the list L_1 of $\mathbf{L}_n(I)$, the list L_2 of $\mathbf{L}_n(J)$, 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 (Property 2). 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 D and as on D' (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 5.21. □

We can now prove:

Theorem 5.22. *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 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 $\mathbb{U}$ 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:

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

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:

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

where I^c denotes $\{1,\ldots,n\} - I$. By $D(I,J)$, 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, $(\pi(I) \cap J) \cup (\pi(I)^c \cap J^c) = \varnothing$. Therefore, for each I, the query RST returns *false* on database $D(I, \pi(I)^c)$, which we will denote by $D(I)$ for short. Furthermore, we observe:

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

Now, for $I \subseteq \{1, \ldots, n\}$, consider the list database $\mathbf{L}_n(I)$ 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 databases is a binary (n, v)-collection with respect to (R, T). (In Definition 5.20, take $x_i = a_i$, $x'_i = a'_i$, $y_i = b_{\pi i}$, and $y'_i = b'_{\pi i}$.)

Now, we apply Lemma 5.21. We thus obtain $I, J \subseteq \{1, \ldots, n\}$ with $I \neq J$ such that the run of M on the list database $\mathbf{L}$ 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 database $\mathbf{L}_n(I)'$ 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 databases $\mathbf{L}_n(I)'$ and $\mathbf{L}$ contain all possible sorted orders of all relations of $D(I)$ and $D(I, \pi(J)^c)$, respectively. Therefore, if M computes the RST query correctly on sorted inputs, M returns *false* on $\mathbf{L}_n(I)'$ and *true* on $\mathbf{L}$ (cf. $(*)$). The runs of M on both list databases, however, end in the same way. We conclude that M can not exist. □

Remark 5.23. **(a)** An analysis of the proof of Lemma 5.21 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 bit strings 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 5.21, 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 5.22 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 5.22 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 bit string, the query RST is easily computed by an $O(n)$-FCM. □

Using Lemma 5.21 we can also show the following strengthening of Theorem 5.4:

Theorem 5.24. *There is no $o(n)$-FCM working on enumerations of unary relations R and S and their reversals, that checks whether $R \cap S \neq \varnothing$.*

Proof. Let M be an $o(n)$-FCM that checks whether $R \cap S \neq \varnothing$. 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 $\mathbb{U}$.

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

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

where the complements I^c and J^c are taken with respect to $\{1, \ldots, n\}$. By $D(I, J)$, 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 $D(I, I^c)$, which we will denote by $D(I)$ for short. Furthermore, we observe:

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

Now, for $I \subseteq \{1, \ldots, n\}$, consider the list databases $\mathbf{L}_n(I)$ 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 databases is a binary (n, v)-collection with respect to (R, S). (In Definition 5.20, take $x_i = y_i = a_i$, $x'_i = y'_i = a'_i$.)

Now, we apply Lemma 5.21. We thus obtain $I, J \subseteq \{1, \ldots, n\}$ with $I \neq J$ such that the run of M on the list database $\mathbf{L}$ 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 database $\mathbf{L}_n(I)'$ containing the lists in $\mathbf{L}_n(I)$ and their reversals. If M computes the query $R \cap S \neq \varnothing?$ correctly, M returns *false* on $\mathbf{L}_n(I)'$ and *true* on $\mathbf{L}$ (cf. $(*)$). The runs of M on both list databases, however, end in the same way. We conclude that M can not exist. □

Note that Theorems 5.22 and 5.24 are valid for arbitrary background structures.

5.6 Concluding remarks

A natural question arising from Corollary 5.9 is whether finite cursor machines with sorting are capable of computing relational algebra queries *beyond* the semijoin algebra. The answer is affirmative:

Proposition 5.25. *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 $\mathsf{sort}_{(1,2),(\prime,\prime)}(R)$ and $\mathsf{sort}_{(2,1),(\prime,\prime)}(R)$.*

Proof. The list $\mathsf{sort}_{(1,2),(\prime,\prime)}(R)$ 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 $\mathsf{sort}_{(2,1),(\prime,\prime)}(R)$, we check this for the first subset in the list. Then, using two cursors over $\mathsf{sort}_{(1,2),(\prime,\prime)}(R)$, we check whether the second subset equals the first, the third equals the second, and so on. □

Note that, using an Ehrenfeucht-game argument (see Section 7.2), one can indeed prove that the query from Proposition 5.25 is not expressible in the semijoin algebra.

We have not been able to solve the following:

Open Problem 5.26. 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?

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 5.27. *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 [36]).
(b): A finite cursor machine that solves this query can proceed as follows: The first check follows by Proposition 5.25; 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 5.18: 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?

6

Streaming

In this chapter, we offer a theoretical framework that attempts to clarify various philosophical questions about stream queries. For instance, if streams are thought of as infinite, and arbitrary queries are modeled as functions from streams to streams, what does it mean for a query to be computable? Is computability the same concept as continuity? What is the precise connection between continuity and monotonicity? Can one give a formal definition of what it means for an arbitrary operator to be non-blocking?

Furthermore, we define a concrete computation model for stream queries, called "streaming ASM", and prove impossibility results. Specifically, we focus on bounded memory machines: such machines can only remember a constant number of previously seen stream elements. Bounded memory machines are natural in the context of query processing; for example, any query operator that applies a sliding window (typical in streaming applications) is computable in bounded memory.

6.1 Abstract computability

Basically, we assume a universe $\mathbb{E}$ of data elements. For example, $\mathbb{E}$ could be the universe $\mathbb{U}$ from which tuples and relations are constructed as in the preceding chapters; $\mathbb{E}$ could contain $\mathbb{U}$ together with all tuples over $\mathbb{U}$, ... A *stream* is a possibly infinite sequence of data elements. The set of all streams is denoted by *Stream*, and the set of all finite streams is denoted by *finStream*. Thus $\mathit{finStream} \subseteq \mathit{Stream}$. We denote the i-th element of a stream $\mathbf{s}$ by s_i.

Our model of streams is very abstract and thus very general.

Example 6.1. Consider measurements coming from sensors, where each entry is a pair of the form (i, m) with i a sensor identifier and m a measurement. Suppose, at each discrete time point t (with time points modeled by natural numbers), we collect all

entries that arrived in the interval $(t-1, t]$. Then $\mathbb{E}$ would contain sets of entries as data elements.

In a setting where time points would be more fine-grained, so that at most one entry can arrive per clock tick, $\mathbb{E}$ would contain entries directly as data elements, plus possibly some dummy element to indicate no entry arrived. □

Mathematically, a *stream query* is simply a mapping from *Stream* to *Stream*. Not all such mappings make sense in the streaming context, however. To make formal which queries do make sense, we define the notion of *abstract computability*. Intuitively, a stream query $\mathcal{Q}$ is abstract computable if there exists a function K: *finStream* → *finStream* such that the result of $\mathcal{Q}$ can be obtained by concatenating the results of K applied to larger and larger prefixes of the input.

Formally, for any K as above, we define the function

$$\mathrm{Repeat}(K)\colon \mathbf{s} \mapsto \bigodot_{k=0}^{\mathrm{size}(\mathbf{s})} K(\mathbf{s}^{\leqslant k}) \quad \text{of type} \quad \mathit{Stream} \to \mathit{Stream},$$

where $\mathbf{s}^{\leqslant k}$ is the prefix of $\mathbf{s}$ of length k, and $\mathrm{size}(\mathbf{s})$ is the length of $\mathbf{s}$ in case $\mathbf{s}$ is finite, and ∞ in case $\mathbf{s}$ is infinite (in which case the index k ranges over all natural numbers). Here $\bigodot$ denotes concatenation. We now define:

Definition 6.2. A query $\mathcal{Q}$: *Stream* → *Stream* is abstract computable if there exists a function K such that $\mathcal{Q} = \mathrm{Repeat}(K)$. We call K a kernel for Q.

The following example shows an abstract computable stream query:

Example 6.3. Let $\mathcal{Q}$ be the *running average* query, defined on streams of natural numbers and returning at each step the average value of the numbers arrived so far. The function returning $(\sum u_i)/n$ on input stream $u_1 \dots u_n$ (and returning the empty stream when the input is the empty stream) is a kernel for $\mathcal{Q}$. □

In connection to finite streams, we make the following two important observations:

1. The answer to an abstract computable query on an infinite stream can be finite.

   Example 6.4. Consider the query $\mathcal{Q}$ that returns all elements in the input stream satisfying a certain predicate P. On a stream with only a finite number of elements satisfying P, the result of $\mathcal{Q}$ will be finite. Note that this query $\mathcal{Q}$ indeed has a kernel: for example the function K that given a finite stream, returns its last element if it satisfies P, and returns the empty stream otherwise, is a kernel for $\mathcal{Q}$. □

2. *The answer to an abstract computable query on a finite stream must be finite.* Indeed, the result of K is always a finite stream and on a finite input stream, K is applied only a finite number of times. So, queries transforming finite streams into infinite streams will never be computable in our model. This is not a problem since our model is primarily meant to capture input-data-driven computations.

We note:

Proposition 6.5. *Abstract computable stream queries are closed under composition.*

Proof. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be abstract computable by F_1 and F_2, respectively. Then, the query $\mathcal{Q} = \mathcal{Q}_2 \circ \mathcal{Q}_1$ is abstract computable by the function F that maps a finite stream $\mathbf{s}u$ to

$$\bigodot_{k=1}^{\text{length}(F_1(\mathbf{s}u))} F_2(Q_1(\mathbf{s})F_1(\mathbf{s}u)^{\leqslant k}),$$

and that maps the empty stream to

$$F_2(()) \odot \bigodot_{k=1}^{\text{length}(F_1(()))} F_2(F_1(())^{\leqslant k}).$$

□

6.2 Continuity

We will now see that abstract computability and continuity of stream queries coincide.

Recall from elementary calculus [8] that a real function $f\colon \mathbb{R} \to \mathbb{R}$ is called continuous if for all $x \in \mathbb{R}$, for every neighborhood around $f(x)$, there exists a neighborhood around x that is completely mapped into the neighborhood of $f(x)$. In order to generalize this definition of continuity to stream queries, we must first agree on a definition of neighborhood of a stream $\mathbf{s}$. In other words, we need to define a suitable topology on streams.

For infinite streams, there is a standard topology, known from computable analysis [60], called the Cantor topology. This topology arises from the following metric (distance function) on infinite streams:

$$d(\mathbf{s}, \mathbf{s}') = \begin{cases} 0 & \text{if } \mathbf{s} = \mathbf{s}', \\ 2^{-n} & \text{if } \mathbf{s} \neq \mathbf{s}' \text{ and } n = \min\{i \mid s_i \neq s'_i\}. \end{cases}$$

According to this topology, open balls around an infinite stream $\mathbf{s}$ are sets of the form $\mathbf{B}(\mathbf{p})$, with $\mathbf{p}$ some finite prefix of $\mathbf{s}$, defined as follows:

$$\mathbf{B}(\mathbf{p}) = \{\mathbf{s}' \text{ infinite stream} \mid \mathbf{p} \text{ is a prefix of } \mathbf{s}'\}.$$

Here, we generalize this notion of open ball to the setting of both finite and infinite streams, as follows:

Definition 6.6. Let $\mathbf{p} \in \mathit{finStream}$. Then

$$\mathbf{B}(\mathbf{p}) := \{\mathbf{s}' \in \mathit{Stream} \mid \mathbf{p} \text{ is a prefix of } \mathbf{s}'\}.$$

Any set of the form $\mathbf{B}(\mathbf{p})$, for some $\mathbf{p} \in \mathit{finStream}$, is called an *open ball*. Elements of $\mathbf{B}(\mathbf{p})$ are called continuations of $\mathbf{p}$.

This notion of open ball gives rise to a topology on streams, and the notion of continuity then amounts to the following:

Definition 6.7. $\mathcal{Q}$: *Stream* $\to$ *Stream* is *continuous* if for every open ball $\mathbf{B}$, the pre-image $\mathcal{Q}^{-1}(\mathbf{B})$ is a union (possibly infinite) of open balls.

Remark 6.8. The Cantor metric described above has only been defined on infinite streams. One may wonder whether the topology on *Stream* given by Definition 6.6 can be given by some metric of that sort but applicable to finite as well as infinite streams. The answer is negative: metrizable topologies must be Hausdorff, and our topology is not. Indeed, an infinite stream $\mathbf{q}$ and a finite prefix $\mathbf{p}$ of $\mathbf{q}$ can not be separated as each open ball containing $\mathbf{p}$ contains $\mathbf{q}$. For basic background on topology, we refer to Hocking and Young [38]. □

Theorem 6.9. *Let $\mathcal{Q}$ be a stream query mapping finite inputs to finite outputs. Then $\mathcal{Q}$ is abstract computable if and only if $\mathcal{Q}$ is continuous.*

Proof. For the only-if direction let K be a kernel for $\mathcal{Q}$, i.e., $\mathcal{Q} = \mathrm{Repeat}(K)$. Consider $\mathbf{X} := \mathbf{B}(\mathbf{p})$. Let $\mathbf{s}$ be a stream in $\mathcal{Q}^{-1}(\mathbf{X})$. Then, from some natural number ℓ on, we know that $\bigodot_{k=0}^{\ell} K(\mathbf{s}^{\leqslant k})$ starts with $\mathbf{p}$. Consider then the open ball $\mathbf{B}(s_1 \dots s_\ell)$. Every $\mathbf{s}' \in \mathbf{B}(s_1 \dots s_\ell)$ is mapped into $\mathbf{X}$. Indeed,

$$\mathcal{Q}(\mathbf{s}') = \bigodot_{k=0}^{\ell} K(\mathbf{s}^{\leqslant k}) \odot \bigodot_{k=\ell+1}^{\mathrm{size}(\mathbf{s})} K(\mathbf{s}'^{\leqslant k})$$

clearly starts with $\mathbf{p}$. Thus, $\mathbf{s} \in \mathbf{B}(s_1 \dots s_\ell) \subseteq \mathcal{Q}^{-1}(\mathbf{X})$, as desired.

For the if-direction, we define a kernel K for Q as follows. $K(()) := \mathcal{Q}(())$, and $K(\mathbf{s}u) := \mathcal{Q}(\mathbf{s}u) - \mathcal{Q}(\mathbf{s})$, where the difference is to be interpreted as removing a prefix, so that $\mathcal{Q}(\mathbf{s}u) = \mathcal{Q}(\mathbf{s}) \odot K(\mathbf{s}u)$. Note that $\mathcal{Q}(\mathbf{s})$ and $\mathcal{Q}(\mathbf{s}u)$ are both finite.

For K to be well-defined, we must show that $\mathcal{Q}(\mathbf{s})$ is indeed a prefix of $\mathcal{Q}(\mathbf{s}u)$. Consider $\mathbf{X} = \mathcal{Q}^{-1}(\mathbf{B}(\mathcal{Q}(\mathbf{s})))$. By continuity, $\mathbf{X}$ is a union of open balls. Thus, there must be an open ball $\mathbf{B}(\mathbf{p})$ with $\mathbf{s} \in \mathbf{B}(\mathbf{p}) \subseteq \mathbf{X}$. Clearly, $\mathbf{p}$ must be a prefix of $\mathbf{s}$. But then also $\mathbf{s}u \in \mathbf{B}(\mathbf{p}) \subseteq \mathbf{X}$, and therefore $\mathcal{Q}(\mathbf{s}u) \in \mathbf{B}(\mathcal{Q}(\mathbf{s}))$. This means that $\mathcal{Q}(\mathbf{s})$ is a prefix of $\mathcal{Q}(\mathbf{s}u)$.

We now show that $\mathrm{Repeat}(K) = \mathcal{Q}$ by showing that they have the same prefixes. By construction, $\mathrm{Repeat}(K)$ coincides with $\mathcal{Q}$ on finite streams. Let $\mathbf{s} = s_1 s_2 \dots$ be an infinite stream and let $v_1 \dots v_j$ be an arbitrary prefix of $\mathrm{Repeat}(K)(\mathbf{s})$. Let i be the smallest natural number such that $v_1 \dots v_j$ is a prefix of $\mathrm{Repeat}(K)(s_1 \dots s_i) = \mathcal{Q}(s_1 \dots s_i)$. Since $\mathcal{Q}(\mathbf{s}) \in \mathbf{B}(\mathcal{Q}(s_1 \dots s_i))$, we have $v_1 \dots v_j$ also as a prefix of $\mathcal{Q}(\mathbf{s})$. We conclude that every prefix of $\mathrm{Repeat}(K)(\mathbf{s})$ is also a prefix of $\mathcal{Q}(\mathbf{s})$.

For the other direction, let $v_1 \dots v_j$ be an arbitrary prefix of $\mathcal{Q}(\mathbf{s})$. By continuity, $v_1 \dots v_j$ is also a prefix of $\mathcal{Q}(s_1 \dots s_i)$ for some i. Since $\mathrm{Repeat}(K)(s_1 \dots s_i) = \mathcal{Q}(s_1 \dots s_i)$, we have $v_1 \dots v_j$ also as a prefix of $\mathrm{Repeat}(K)(s_1 \dots s_i)$, which by construction is itself a prefix of $\mathrm{Repeat}(K)(\mathbf{s})$, as desired. □

Theorem 6.9 can be used to prove that there are simple stream queries that are not abstract computable.

Example 6.10. Consider the following query CHECK. Let $a, b \in \mathbb{E}$ and let $\mathbf{s}$ be a stream over $\mathbb{E}$. Then $\textsc{check}(\mathbf{s})$ is the stream (a) if b does not occur in $\mathbf{s}$; otherwise, $\textsc{check}(\mathbf{s})$ is the empty stream $()$. This query is not abstract computable; we prove that CHECK

is not continuous. Consider the open ball $\mathbf{B}(a)$. Clearly, the empty stream () is in $\textsc{check}^{-1}(\mathbf{B}(a))$. The only open ball that contains the empty stream is $\mathbf{B}(())$. This open ball, however, is not included into $\textsc{check}^{-1}(\mathbf{B}(a))$. Indeed, $(b) \in \mathbf{B}(())$, but $\textsc{check}(b) = () \notin \mathbf{B}(a)$. □

Remark 6.11. In connection to Theorem 6.9 we remark the following:

1. Suppose we would have extended the Cantor metric to finite (as well as infinite) streams in the obvious manner; in particular, if $\mathbf{s}$ is a finite prefix of $\mathbf{s}'$, but $\mathbf{s} \neq \mathbf{s}'$, then we define $d(\mathbf{s}, \mathbf{s}') = 2^{-(n+1)}$ with n the length of $\mathbf{s}$. In the resulting topology, abstract computable queries need no longer be continuous. A simple example is provided by the query $\mathcal{Q}$ from Example 6.4. Let $\mathbb{E} := \{a, b\}$ and let P be true of a and false of b. Consider the open ball $\mathbf{B}$ containing only the empty stream (). Then $\mathcal{Q}$ maps the infinite stream $\mathbf{b}$ containing only b's into $\mathbf{B}$. Any open ball $\mathbf{B}(\mathbf{p})$ around $\mathbf{b}$, however, contains the stream $\mathbf{p}a$ which is not in $\mathcal{Q}^{-1}(\mathbf{B})$. Thus, $\mathcal{Q}$ is not continuous.

2. The qualification in Theorem 6.9 that $\mathcal{Q}$ must map finite inputs to finite outputs is important for the if-direction. Indeed, any constant query, that always outputs some fixed infinite stream, is continuous, but not abstract computable (precisely because it maps finite to infinite). □

6.3 The finite case

Considering only finite streams makes the situation simpler. Define a finite stream query as a mapping from *finStream* to *finStream*. Define abstract computability of finite stream queries in the same way as for queries on *Stream*, and consider the topology on *finStream* induced by the topology on *Stream*, i.e., the open balls are now *finite* continuations of finite streams. We will use the notation $\mathbf{B}_{\text{fin}}(\mathbf{p})$ to denote the set of all finite continuations of the finite stream $\mathbf{p}$. We then indeed have:

Proposition 6.12. *A finite stream query is abstract computable if and only if it is continuous.*

In the finite case, there is also a third equivalent notion: monotonicity. A query $\mathcal{Q}$: *finStream* $\rightarrow$ *finStream* is called monotone if for all $\mathbf{s}, \mathbf{s}' \in$ *finStream*, $\mathbf{s} \sqsubseteq \mathbf{s}'$ implies $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$, where $\sqsubseteq$ denotes the "prefix of" relation.

Proposition 6.13. *A finite stream query is continuous if and only if it is monotone.*

Proof. For the if-direction let $\mathcal{Q}$: *finStream* $\rightarrow$ *finStream* be monotone. Consider $\mathbf{X} := \mathbf{B}_{\text{fin}}(\mathbf{p})$. Let $\mathbf{s}$ be a stream in $\mathcal{Q}^{-1}(\mathbf{X})$. Then, $\mathbf{s} \in \mathbf{B}_{\text{fin}}(\mathbf{s}) \subseteq \mathcal{Q}^{-1}(\mathbf{X})$. Indeed, $\mathbf{s}' \in \mathbf{B}_{\text{fin}}(\mathbf{s})$ implies $\mathbf{s} \sqsubseteq \mathbf{s}'$, which by monotonicity implies $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$. As $\mathcal{Q}(\mathbf{s})$ has $\mathbf{p}$ as a prefix, $\mathcal{Q}(\mathbf{s}')$ has $\mathbf{p}$ as a prefix too and thus $\mathcal{Q}(\mathbf{s}') \in \mathbf{X}$.

The only-if direction is proved by the argument already used in the proof of the if-direction of Theorem 6.9, where we showed that K is well-defined. Concretely, let $\mathcal{Q}$: *finStream* $\rightarrow$ *finStream* be continuous. Let $\mathbf{s} \sqsubseteq \mathbf{s}'$. Consider $\mathbf{X} := \mathcal{Q}^{-1}(\mathbf{B}_{\text{fin}}(\mathcal{Q}(\mathbf{s})))$. By continuity, $\mathbf{X}$ is a union of open balls. Thus, there must be an open ball $\mathbf{B}_{\text{fin}}(\mathbf{p})$ with $\mathbf{s} \in \mathbf{B}_{\text{fin}}(\mathbf{p}) \subseteq \mathbf{X}$. Clearly, $\mathbf{p}$ must be a prefix of $\mathbf{s}$. But

then also $\mathbf{s}' \in \mathbf{B}_{\text{fin}}(\mathbf{p}) \subseteq \mathbf{X}$, and therefore $\mathcal{Q}(\mathbf{s}') \in \mathbf{B}_{\text{fin}}(\mathcal{Q}(\mathbf{s}))$. This means that $\mathcal{Q}(\mathbf{s}) \sqsubseteq \mathcal{Q}(\mathbf{s}')$. □

As a corollary we obtain the following equivalence already noted by Law, Wang and Zaniolo (LWZ), who referred to our notion of abstract computability as computability by a "nonblocking" operator:

Corollary 6.14 ([41])**.** *Let $\mathcal{Q}$ be a finite stream query. $\mathcal{Q}$ is computable by a non-blocking operator if and only if $\mathcal{Q}$ is monotone.*

The proof given by LWZ is slightly confusing. Their formalism is based on a notion of queries on finite streams that are computable by (not necessarily non-blocking) "operators". They fail to mention, however, that *any* query on finite streams is computable by such an operator.

6.4 Time

In some applications, the output stream is synchronized with the input stream. In such cases, we need an additional requirement on stream queries beyond mere abstract computability.

Example 6.15. Consider the following instance of the query from Example 6.4: the input is a stream of numbers (e.g., sensor readings) and the output consists of all readings below a certain threshold, say 0. In an "untimed" setting, where the original time points of the output readings are not required by the client of the query, we can simply formalize this stream query as being abstract computable with kernel function K_0 with $K_0(()) = ()$, and

$$K_0(\mathbf{s}u) = \begin{cases} u & \text{if } u < 0 \\ () & \text{otherwise.} \end{cases}$$

On the other hand, in a "timed" setting stream positions in the output are supposed to be synchronized with stream positions in the input [7, 6]. In that case, the above formalization is inadequate, because, the 5th element of the output may well be, say, the 10th element of the input!

A more proper computation would be given by the function K_1 with again $K_1(()) = ()$, and now

$$K_1(\mathbf{s}u) = \begin{cases} u & \text{if } u < 0 \\ \text{NULL} & \text{otherwise.} \end{cases}$$

where NULL is an explicitly visible element denoting that the reading at this time point was not below 0. □

The above discussion motivates:

Definition 6.16. A stream query $\mathcal{Q}$ is *synchronous abstract computable* (SAC) if $\mathcal{Q} = \text{Repeat}(K)$ for some kernel $K\colon \mathit{finStream} \to \mathit{finStream}$ such that $K(()) = ()$ and every other $K(\mathbf{s})$ is of length one. We will call such kernel K a *length-one kernel*.

SAC stream queries can be characterized by means of non-predicting queries. Here and below, $\mathbb{N}_0$ stands for the set of natural numbers without zero.

Definition 6.17. A stream query $\mathcal{Q}$ is *non-predicting* if for all streams $\mathbf{s}$ and $\mathbf{s}'$ and for all $t \in \mathbb{N}_0$ such that $\mathbf{s}^{\leqslant t} = (\mathbf{s}')^{\leqslant t}$, we have $\mathcal{Q}(\mathbf{s})_t = \mathcal{Q}(\mathbf{s}')_t$.

We note that non-predicting is part of the definition of "stream operator" by Arasu, Babu and Widom [7, 6].

Proposition 6.18. *A stream query is SAC if and only if it is non-predicting.*

Proof. Let K be a length-one kernel for stream query $\mathcal{Q}$. Let $\mathbf{s}, \mathbf{s}' \in \mathit{Stream}$ and $t \in \mathbb{N}_0$ such that $\mathbf{s}^{\leqslant t} = (\mathbf{s}')^{\leqslant t}$. Then

$$\mathcal{Q}(\mathbf{s})_t = K(\mathbf{s}^{\leqslant t}) = K((\mathbf{s}')^{\leqslant t}) = Q(\mathbf{s}')_t$$

and thus $\mathcal{Q}$ is non-predicting.

For the "if" direction, let $\mathcal{Q}$ be non-predicting. For each finite stream $\mathbf{p}$ of length t, define $\pi(\mathbf{p})$ as the infinite stream with $\pi(\mathbf{p})_i = p_i$ for $i \leqslant t$ and with $\pi(\mathbf{p})_i = p_t$ for $i > t$. Then the following function K is a length-one kernel for $\mathcal{Q}$. If $\mathbf{p}$ is a finite stream of length t then $K(\mathbf{p}) := \mathcal{Q}(\pi(\mathbf{p}))_t$.

Furthermore, for each stream $\mathbf{s}$ and any time instant t, define $\pi'(\mathbf{s}, t)$ as the infinite stream with $\pi'(\mathbf{s}, t)_i = s_i$ for $i \leqslant t$ and with $\pi'(\mathbf{s}, t)_i = s_t$ for $i > t$. We now prove that K is indeed as desired. Let $\mathbf{s}$ be a stream and let t be a time instant. Then

$$\mathcal{Q}(\mathbf{s})_t = \mathcal{Q}(\pi'(\mathbf{s}, t))_t = \mathcal{Q}(\pi(\mathbf{s}^{\leqslant t}))_t = K(\mathbf{s}^{\leqslant t}).$$

Here, the first equality follows from the fact that $\mathcal{Q}$ is non-predicting; the second equality follows from the fact the definition of $\pi'(s, t)$; and the third equality follows from the definition of K. □

We also have:

Proposition 6.19. *SAC stream queries are closed under composition.*

Proof. Let $\mathcal{Q}_1$ and $\mathcal{Q}_2$ be abstract computable by F_1 and F_2, respectively. It is easy to verify that given that F_1 and F_2 satisfy the properties in Definition 6.16, the function F constructed in the proof of Proposition 6.5 also satisfies these properties and makes $\mathcal{Q}_2 \circ \mathcal{Q}_1$ synchronous abstract computable. □

6.5 Complexity limitations

The definition of abstract computability does not impose any restriction on K: the function is not even required to be computable, neither in the classical sense nor in the sense of TTE. The results in the previous sections are thus very general.

To further study the limitations of streaming applications, however, such restrictions are necessary. Concretely, for a class $\mathcal{C}$ of functions from *finStream* to *finStream*, we say that a query $\mathcal{Q}$: *Stream* $\to$ *stream* is *abstract computable modulo* $\mathcal{C}$ if $\mathcal{Q}$ has a kernel K in $\mathcal{C}$. The class $\mathcal{C}$ could for example be the class of functions computable

in the classical sense or in the sense of TTE; or—as in the "streaming model of computation" [9]—$\mathcal{C}$ could be the class of functions incrementally computable in polylog space and in polylog time per data element.

In the next section, we will define several classes $\mathcal{C}$ of functions computable by a concrete model based on the Abstract State Machine (ASM) methodology [31], that we will call "streaming ASM" (sASM). We will study abstract computability modulo the classes $\mathcal{C}$ obtained by altering the computation power of the model.

6.6 Streaming ASMs

An abstract state machine (ASM) is a transition system whose states are many-sorted first-order structures. Transitions change the interpretation of some of the function and relation symbols—those in the *dynamic* part of the vocabulary—and leave the remaining symbols—those in the *static* part of the vocabulary—unchanged. The part of the structure that is never changed during state transitions, i.e., the structure over the static part of the vocabulary, is typically called the *background structure*. Transitions are described by simple rules that produce state updates which are "fired" simultaneously (if they are inconsistent, no update is carried out). A crucial property of the sequential ASM model 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 [31].

We now describe the streaming abstract state machine (sASM) model.

The states: The base set of any state, i.e., the universe of the structure in the sense of logic, contains at least our universe $\mathbb{E}$ of data elements. We assume that $\mathbb{E}$ contains an element $\perp$.

The static functions and predicates on the base set include, but are not limited to, the functions and predicates defined on $\mathbb{E}$.

Each state of an sASM contains a finite number of dynamic functions on the base set. There are always the nullary dynamic function *in* and a number of nullary dynamic functions, called output registers, denoted by *out*, possibly with subscripts. The output registers and *in* take values in $\mathbb{E}$.

The names of the static and dynamic functions and predicates are collected in a vocabulary.

The program: A program for an sASM is a basic sequential program in the sense of ASM theory. Concretely, a basic update rule has the form: $f(t_1, \ldots, t_n) := t_0$ where f is a function name and $t_0, \ldots, t_n$ are terms in the vocabulary. To fire the basic update rule at a state $\mathcal{A}$, evaluate the terms $t_0, \ldots, t_n$ in $\mathcal{A}$ to elements $a_0, \ldots, a_n$ in the base set and then change the interpretation of f in $(a_1, \ldots, a_n)$ to a_0.

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 this: Fire rules $r_1, \ldots, r_m$ in parallel; if they are inconsistent then do nothing.

Furthermore, if r_1 and r_2 are rules and φ is a quantifier-free formula in the vocabulary, then $\mathsf{if}\ \varphi\ \mathsf{then}\ r_1\ \mathsf{else}\ r_2\ \mathsf{endif}$ is also a rule. The semantics is obvious.

Now, an sASM program is just a single rule.

The run and the output: An sASM M that is set to work on a finite stream $\mathbf{s}$ starts in the state where all dynamic functions have the interpretation $\perp$, except for the function *in*: In the initial state, the function *in* contains the first element of the stream $\mathbf{s}$.

The run of M on $\mathbf{s}$ is the sequence of states obtained as follows: start from the initial state and fire (the rule of) M's program, in each step interpreting the function *in* as the next element in $\mathbf{s}$. The sASM halts when the end of $\mathbf{s}$ is reached. The interpretation of *in* is dynamic but it is controlled by the environment rather than by the machine; *in* is an *external* function.

We define the *final output of M on a finite stream* $\mathbf{s}$ as the stream obtained by concatenating the interpretations of the output registers in some predefined order when M has halted, and where $\perp$-elements are disregarded.

We now say that an sASM M computes a function $K\colon \mathit{finStream} \to \mathit{finStream}$ (meant as a kernel for a stream query) if for all finite streams $\mathbf{s}$, the final output of M on $\mathbf{s}$ equals $K(\mathbf{s})$. By K_M we denote the function K computed by M.

It is important to note that the final output of an sASM M on a stream $s_1 \dots s_{n+1}$ can be simply obtained by running M on the input $s_1 \dots s_n$ first, and then making one final step upon reading s_{n+1}. Consequently, on any stream $\mathbf{s}$ (finite or infinite), we can compute $\mathrm{Repeat}(K_M)(\mathbf{s})$ simply by continuously running M on $\mathbf{s}$, at each step producing the current output. We refer to $\mathrm{Repeat}(K_M)$ as the *stream query computed by M*.

Example 6.20. Recall Example 6.1. In the setting where $\mathbb{E}$ contains sets of entries, there could for example be a function defined on $\mathbb{E}$ that given a set of entries, returns the set of sensor identifiers that measured an alarmingly high value.

In the setting where $\mathbb{E}$ contains entries directly, there could for example be a predicate defined on $\mathbb{E}$ that checks whether an entry has an alarmingly high measurement and a function that given an entry, returns the sensor identifier of the entry. □

Example 6.21. Consider the sliding window join between two streams of tuples of natural numbers over the attributes $\{A, B\}$ and $\{C, D\}$, where the join condition is $B = C$. The output tuples have attributes $\{A, B, D\}$. The universe $\mathbb{E}$ then contains $\perp$, Tuple_{AB}, Tuple_{CD}, and Tuple_{ABD}, with Tuple_{AB} the set of tuples over the attributes $\{A, B\}$, and similarly for Tuple_{CD} and Tuple_{ABD}. The function $\mathit{join}_{B=C}\colon \mathit{Tuple}_{AB} \times \mathit{Tuple}_{CD} \to \mathit{Tuple}_{ABD}$ checks whether two tuples join on their B- and C-attributes and returns the joined tuple; the result is $\perp$ if the tuples do not join.

The output of the sliding window join depends on two streams, whereas streaming ASMs work on a *single* stream. Moreover, the output depends on the particular interleaving in which the streams arrive. By choosing an appropriate universe $\mathbb{E}$, however, we can represent the two input streams and their interleaving as a single stream.

Concretely, we extend the universe $\mathbb{E}$ with the set *TaggedTuple* of elements of the form $\langle \mathtt{r}\colon u\rangle$ and $\langle \mathtt{s}\colon v\rangle$ with $u \in \mathit{Tuple}_{AB}$ and $v \in \mathit{Tuple}_{CD}$. A tagged tuple encodes an element and its origin. For example, the stream of tagged tuples

$$\langle \mathtt{r}\colon(1,2)\rangle\langle \mathtt{s}\colon(2,3)\rangle\langle \mathtt{s}\colon(3,4)\rangle\dots$$

is a representation of the interleaving of the tuple (1,2) arriving in the first stream, followed by the tuples (2,3) and (3,4) arriving in the second stream, and so on. Furthermore, we add the predicates R and S to the universe $\mathbb{E}$ to test whether an element is of the form $\langle \mathtt{r}:u \rangle$ or $\langle \mathtt{s}:v \rangle$, respectively. Finally, we add a function $\mathit{strip}\colon \mathit{TaggedTuple} \to \mathit{Tuple}_{AB} \cup \mathit{Tuple}_{CD}$ that removes the tag of a tagged tuple. Static functions return $\bot$ when one of the arguments is $\bot$.

Assume for simplicity that the window size is 2. We then equip the sASM with 4 nullary dynamic functions reg_i^R, and reg_i^S for $i = 1, 2$. The following is now a program for an sASM computing the sliding window join described above.

```
par
    if R(in) then
        par
            reg_1^R = in
            reg_2^R = reg_1^R
            out_1 = join_{B=C}(strip(in), strip(reg_1^S))
            out_2 = join_{B=C}(strip(in), strip(reg_2^S))
        endpar
    endif
    if S(in) then
        par
            reg_1^S = in
            reg_2^S = reg_1^S
            out_1 = join_{B=C}(strip(reg_1^R), strip(in))
            out_2 = join_{B=C}(strip(reg_2^R), strip(in))
        endpar
    endif
endpar
```

□

6.7 Bounded-memory and $o(n)$-bit string sASMs

Due to the extreme generality of the ASM model, one should not expect that restricting attention to stream queries that are computable by an sASM would imply any limitation. Indeed, the only restriction that comes from our sASM model is that at each step in the computation of the stream query, only a constant number of elements can be output. More concretely, since the background structure of an sASM could, a priori, be anything, we have the following proposition and corollary (which in itself are philosophically entirely uninteresting):

Proposition 6.22. *Let k be a fixed natural number and let $K\colon \mathit{finStream} \to \mathit{finStream}$ be any kernel function such that the length of $K(\mathbf{s})$, for any finite stream $\mathbf{s}$, is at most k. Then the stream query* Repeat(K) *is computable by some sASM.*

sketch. It is an easy matter for an sASM to compute Repeat(K) if it has 1) a background structure containing a) the set of all finite streams *finStream*, b) the append function of sort $\mathit{finStream} \times \mathbb{E} \to \mathit{finStream}$, c) the function K, and d) functions $\mathit{element}_i$ for $i = 1, \ldots, k$ to extract elements out of a finite stream; and 2) a nullary

dynamic function s containing at each step the part of the stream that has already arrived.

At each step, the sASM uses the append function to update the dynamic function s; it applies K to the stream s; and it uses the extraction functions $element_i$ to update the output registers. □

Corollary 6.23. *Every SAC query is abstract computable by an sASM.*

In order to formulate a relevant complexity limitation on stream queries, we propose "bounded-memory sASMs".

Definition 6.24. A bounded-memory sASM is an sASM with the following restrictions: 1) no output register can ever be used as an argument to a function; 2) all dynamic functions are nullary; and 3) non-nullary (static) functions can only be applied in rules of the form $out := t_0$, with out an output register and t_0 a term over the vocabulary.

Example 6.25. The sASM computing the sliding window join in Example 6.21 is a bounded-memory sASM. The obvious sASM for computing the running average query from Example 6.3, however, is not bounded-memory (but see later, when we introduce bit string sASMs). □

Every CQL-query where a finite window is applied to the input streams ([6]) is computable by a bounded-memory sASM. Indeed, let $\mathcal{Q}$ be such a CQL-query. Then, $\mathcal{Q} = \text{Repeat}(K_M)$, where M is the following sASM. For each window of $\mathcal{Q}$ of size n, the sASM M has n dynamic constants. When M receives a new input element, say with tag $\langle \mathtt{r}: \rangle$, the sASM simulates the sliding of the window(s) on input stream $\mathbf{r}$ by updating the corresponding dynamic constants accordingly. In each step, the output is computed in a brute-force way. This technique was already illustrated in Example 6.21.

Moreover, every duplicate-eliminating SPJ-query computable in bounded memory in the sense defined by Arasu et al. is computable by a bounded-memory sASM [5].

Bounded-memory sASMs also have some limitations: even the very simple stream query that checks whether two streams intersect, is not computable by a bounded-memory sASM. We extend the universe $\mathbb{E}$ with the set *TaggedElement* of elements of the form $\langle \mathtt{r}: u \rangle$ and $\langle \mathtt{s}: u \rangle$ with $u \in \mathbb{E}$. A stream over *TaggedElement* then represents the interleaving of two streams over $\mathbb{E}$ (see Example 6.21). Furthermore, we extend $\mathbb{E}$ with the boolean values `true` and `false`. The query INTERSECT is defined on streams over $\mathbb{E}$ and checks whether a common element has been seen in the interleaved streams. Concretely, the result of INTERSECT on a stream $\mathbf{s}$ over $\mathbb{E}$ is the stream $\mathbf{s}'$ such that the n-th element of $\mathbf{s}'$ is `true` if and only if for some $i, j \in \mathbb{N}_0$ with $i, j < n$ and for some $u \in \mathbb{E}$, we have $s_i = \langle \mathtt{r}: u \rangle$ and $s_j = \langle \mathtt{s}: u \rangle$.

Proposition 6.26. INTERSECT *is not computable by a bounded-memory sASM.*

Proof. Let M be a bounded-memory sASM such that INTERSECT equals $\text{Repeat}(K_M)$.

Let Γ be the set of predicates of M. Then for each predicate $p \in \Gamma$ of arity k and for each k-sequence α of elements in $\{\mathbf{r}, \mathbf{s}\}$, define the predicate p^α on $\mathbb{E}$ to be true of a tuple $(u_1, \ldots, u_k)$ iff p is true of $(\langle \alpha_1 : u_1 \rangle, \ldots, \langle \alpha_k : u_k \rangle)$. Let $\Gamma' := \{p^\alpha \mid p \in \Gamma \text{ and } \alpha \in \{\mathbf{r}, \mathbf{s}\}^k \text{ where } k = \text{arity}(p)\}$.

Without loss of generality, we assume that $\mathbb{E}$ is totally ordered by a predicate $<$. Using Ramsey's theorem, we can find an infinite set $V \subseteq \mathbb{E}$ over which the truth of the predicates in Γ' on tuples of elements in $\mathbb{E}$ only depends on the way these data elements compare w.r.t. $<$ (details on this can be found, e.g., in Libkin's textbook [46, Section 13.3]). Now choose $2n$ elements in V, for n large enough, satisfying $v_1 < v'_1 < \cdots < v_n < v'_n$. Let $\mathbf{s}$ be the input stream $\langle \mathtt{r}:v_1\rangle \ldots \langle \mathtt{r}:v_n\rangle$ and consider the run of M on $\mathbf{s}$. After the step where $\langle \mathtt{r}:v_n\rangle$ is processed there will be at least one element $\langle \mathtt{r}:v_\ell\rangle$ that M has not stored in its registers. Then, consider the streams $\mathbf{s}'$ and $\mathbf{s}''$ of length $n+1$ that have $\mathbf{s}$ as a prefix, and with $s'_{n+1} = \langle \mathtt{s}:v_\ell\rangle$ and $s''_{n+1} = \langle \mathtt{s}:v'_\ell\rangle$. The runs of M on $\mathbf{s}'$ and $\mathbf{s}''$ will be identical to the run of M on $\mathbf{s}$ until right after the step where $\langle \mathtt{r}:v_n\rangle$ is processed. In the next step, the machine receives either $\langle \mathtt{s}:v_\ell\rangle$ or $\langle \mathtt{s}:v'_\ell\rangle$. Because v_ℓ and v'_ℓ have the same relative order with respect to the other v-elements, each tuple of elements from the set $\{v_1, \ldots, v_\ell, \ldots, v_m\}$ satisfies the same predicates in Γ' as the tuple obtained by replacing v_ℓ by v'_ℓ. By definition of Γ', also each tuple of elements from the set $\{\langle \mathtt{r}:v_1\rangle, \ldots, \langle \mathtt{s}:v_\ell\rangle, \ldots, \langle \mathtt{r}:v_m\rangle\}$ satisfies the same predicates in Γ as the tuple obtained by replacing $\langle \mathtt{s}:v_\ell\rangle$ by $\langle \mathtt{s}:v'_\ell\rangle$. Therefore, the output of M on $\mathbf{s}'$ will be identical to the output of M on $\mathbf{s}''$. As a consequence, $\text{Repeat}(K_M)(\mathbf{s}')$ and $\text{Repeat}(K_M)(\mathbf{s}'')$ are equal while $\text{INTERSECT}(\mathbf{s}')$ and $\text{INTERSECT}(\mathbf{s}'')$ are different. Thus, M is wrong. □

This result can also be obtained via a reduction from a result on Finite Cursor Machines (FCMs). Indeed, in Chapter 5 we showed that no matter how rich the background is, even an FCM can not check whether two sets intersect using bit string registers of size $o(n)$, where n is the size of the input (Theorem 5.24, page 59).

The proof we gave here is more direct and therefore provides more insight on the limitations of bounded memory stream processing. The reduction argument, however, can easily be generalized to accommodate for bit string registers of size $o(n)$. A *bit string sASM* is an sASM defined as in Definition 6.24 with the following relaxation of restriction 3: non-nullary (static) functions can be used also to update non-output registers, as long as those functions produce bit strings. An $o(n)$-sASM then is a bit string sASM such that on each stream $\mathbf{s}$ and for each step n in the run on $\mathbf{s}$, the sASM stores bit strings of length $o(n)$.

Example 6.27. We can model a version of the running average query (Example 6.3) using $o(n)$-bit string sASMs. Indeed, consider streams of natural numbers such that the value in the n-th position of the stream (for any n) is at most $2^{\text{polylog}(n)}$. Then with a static function from natural numbers to their binary representations, and the addition and division function on binary numbers, we can compute the running average with an $o(n)$-sASM. □

Proposition 6.28. *The query* INTERSECT *is not computable by an $o(n)$-sASM.*

Proof. Let M be an $o(n)$-sASM M working on a stream of tagged elements such that INTERSECT is equal to $\text{Repeat}(K_M)$. From M, we can then construct an $o(n)$-FCM M' working on two lists of elements in $\mathbb{E}$ that checks whether they have a common element. The FCM M' has the same number of bit string registers as M, and has an element register for every dynamic constant of M. For every element in an element register, M' remembers from which input list the element was copied, using

its internal mode. Furthermore, let Γ be the set of predicates of M, including the predicates naturally corresponding to M's boolean output functions. Then the set of predicates of M' is the set Γ' as defined in the proof of Proposition 6.26. Finally, if $\mathcal{F}$ is the set of functions of M, then the set of functions $\mathcal{F}'$ of M' is similarly constructed from $\mathcal{F}$ as Γ' is constructed from Γ.

Consider the input lists R and S. The FCM M' has a single cursor on R and a single cursor on S. Now, M' computes as follows. At each odd step, M' moves its cursor on R to the next element u, updating the (element and bit string) registers as M would do when receiving the element $\langle \mathtt{r}{:}u \rangle$ from the stream. The internal mode is changed so that it contains the origin of each element in the registers. At each even step, M' moves its cursor on S to the next element v, updating the registers as M would do when receiving the element $\langle \mathtt{s}{:}v \rangle$ from the stream. The internal mode is again changed accordingly. M' can simulate this behaviour using the functions in $\mathcal{F}'$, or the predicates in Γ' together with its internal mode. For example, if M applies a predicate p to an element in a dynamic constant *reg* — i.e., an element of the form $\langle \mathtt{r}{:}u \rangle$ or $\langle \mathtt{s}{:}v \rangle$ — the FCM M' would use its internal mode to obtain the origin of the element in the register corresponding to *reg* and then apply the right predicate p^r or p^s to the element in that register — i.e., to u or v. Once M outputs true, M' enters the accept state and halts. As long as M outputs false, M' continues until it has detected the ends of the input lists. In that case, M' enters the reject state and halts. Note that M' can use the predicates corresponding to the boolean functions of M to obtain the output M produces. Because M works correctly, it will also work correctly on this particular interleaving. Therefore, M' correctly checks whether R and S intersect. Hence the contradiction. □

We conclude by pointing out that on finite streams, finite cursor machines are indeed more powerful than bounded-memory sASMs: Consider the query SORT-INTERSECT that given two finite streams A and B, checks if they are both sorted and if so, outputs their intersection; if the inputs are not sorted, SORT-INTERSECT, outputs false. Then,

Proposition 6.29. *The query* SORT-INTERSECT *is computable by an FCM but not by a bounded-memory sASM.*

Proof. An FCM would compute the query SORT-INTERSECT using one cursor on each list to check if they are sorted and another cursor on each list to do a synchronized scan of both list to search for common elements. Inspection of the proof of Proposition 6.26 reveals that a bounded-memory sASM can not even check whether two finite sorted streams intersect. □

6.8 Conclusion

An interesting open problem is to relax the definition of bounded-memory sASM in other ways than with using $o(n)$-length bit strings.

7

The expressive power of the semijoin algebra

In this chapter, we study the expressive power of the semijoin algebra in the presence of arbitrary predicates in the selection and join conditions. Note that the Codd theorem for the semijoin algebra (Chapter 3) only considers equi-semijoins.

The first part of this chapter deals with repetitions and permutations of columns. While in the full relational algebra permuting and repeating of columns does not add expressive power, this is not clear for the semijoin algebra. Indeed, the rewrite rule to replace a permuting or repeating projection in a relational algebra expression with a non-permuting and non-repeating one uses the join operator, that the semijoin algebra lacks. Nevertheless, we show that any semijoin algebra expression can still be simulated by semijoin algebra expressions where no projection operator permutes or repeats columns. The idea is that given an arbitrary expression E, one can produce a set of permutation- and repetition-free expressions that return the relevant values of the output tuples of E, up to certain repetitions and permutations which are produced as a by-product of the translation. In particular, for boolean expressions, there is always a single equivalent boolean expression that is permutation- and repetition-free.

In a second part of this chapter, we define an Ehrenfeucht-Fraïssé game, that characterizes the discerning power of the semijoin algebra in the presence of arbitrary predicates in the selection and join conditions. In Section 3.8, we already remarked that allowing nonequalities in semijoin conditions strictly increases the expressive power of the semijoin algebra. Unfortunately, the increase in expressive power leads to an undecidable satisfiability problem. Nevertheless, it is still interesting to study the expressive power of this more powerful semijoin algebra. Moreover, using the Ehrenfeucht-Fraïssé game as a tool, we will particularly study the expressive power

of $\mathrm{SA}^{\neq}$ and $\mathrm{SA}^{<,<}$.

7.1 Repetitions and permutations of columns

7.1.1 The restrictions SA^{-r} and SA^{-rp}

The restrictions RA^{-r} and SA^{-r} of RA and SA, respectively, are obtained by restricting the projection operator $\pi_{i_1,\ldots,i_k}$ (see Chapter 2) by requiring that the numbers $i_1,\ldots,i_k$ are all different. So, repeating columns in the projection list is not allowed. The restrictions RA^{-rp} and SA^{-rp} are obtained by requiring the projection list $i_1,\ldots,i_k$ to be strictly increasing, i.e., $i_1 < \cdots < i_k$. So, permutations are not allowed. Note that this also excludes repeating columns.

Remark 7.1. Note that the intersection operator $\cap$ is expressible in RA using the projection and the join operator: $R \cap S = \pi_{1,\ldots,\mathrm{arity}(R)}(R \bowtie_\theta S)$, where θ is the formula $\bigwedge_{i=1}^{\mathrm{arity}(R)} x_i = y_i$. By definition of the semijoin operator, the intersection $R \cap S$ is also expressible in SA as $R \ltimes_\theta S$, where θ is as before. □

Example 7.2. Let $\mathbf{S}$ be the schema containing a single binary relation Knows. Then the SA^{-r} (and, hence, RA^{-r}) expression $\mathrm{Knows} \cap \pi_{2,1}(\mathrm{Knows})$ defines all pairs of persons who know each other. The expression is not SA^{-rp} (nor RA^{-rp}), but it can be equivalently written in SA^{-rp} (and RA^{-rp}) as $\mathrm{Knows} \underset{\substack{1=2\\2=1}}{\ltimes} \mathrm{Knows}$ □

7.1.2 Expressive power of SA^{-r} and SA^{-rp}

Here, we show the main result of Section 7.1: allowing permuting and repeating of columns in projections does not add expressive power to the semijoin algebra. Note that this property is clear for the full relational algebra. Indeed, if R is a relation of arity n and $i_1,\ldots,i_k$ are values between 1 and n, then $\pi_{i_1,\ldots,i_k}R$ is equivalent to the RA^{-rp} expression

$$\pi_{f(1),\ldots,f(k)}\Big(\underbrace{((R \underset{\theta}{\bowtie} R) \underset{\theta}{\bowtie} \ldots) \underset{\theta}{\bowtie} R}_{k \text{ times } R}\Big)$$

where $f(j)$ is $(j-1)n + i_j$ and θ is $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$.

Example 7.3. The RA^{-r} expression $\pi_{2,1}(\mathrm{Knows})$ can be expressed in RA^{-rp} as

$$\pi_{2,3}(\mathrm{Knows} \underset{\substack{1=1\\2=2}}{\bowtie} \mathrm{Knows}).$$

□

This trick for RA does not work for SA, where we do not have the full join. Indeed, a projection with repetitions like $\pi_{1,1}R$ cannot be equivalently expressed in SA^{-rp}. The same holds for a nonincreasing projection like $\pi_{2,1}R$. Nevertheless, for any SA expression E that can use projections with arbitrary repetitions and permutations, we can still obtain the tuples returned by E, by means of SA^{-rp} expressions. We formally state and prove this in the following theorem. But first, we introduce a notation: Let f be a function from $\{1,\ldots,m\}$ to $\{1,\ldots,n\}$ and let $\overline{a}$ be an n-tuple. Then $f(\overline{a})$ is the m-tuple $(a_{f(1)},\ldots,a_{f(m)})$.

Theorem 7.4. *Let E be an SA expression of arity n. Then there exists a set P of pairs of the form (F, f), where F is an SA^{-rp} expression and f is a function from $\{1, \ldots, n\}$ to $\{1, \ldots, \ell\}$ with ℓ the arity of F, such that for each database D:*

$$E(D) = \bigcup_{(F,f)\in P} \{f(\overline{a}) \mid \overline{a} \in F(D)\}.$$

Before we prove the theorem, we give an example and make a remark.

Example 7.5. If E is the expression $\pi_{2,1,1}R$, then a set P according to Theorem 7.4 is the singleton $\{(R, f)\}$, where f is the function from $\{1, 2, 3\}$ to $\{1, 2\}$ with $f(1) = 2$ and $f(2) = f(3) = 1$.

If E is the expression $(\pi_{2,1,2}R) \ltimes_{2=1} (\pi_{3,1}S)$, then a set P according to Theorem 7.4 is the singleton $\{(R \ltimes_{1=3} S, f)\}$, where $f(1) = f(3) = 2$ and $f(2) = 1$. □

Remark 7.6. To see why in general P contains more than one element, consider the schema $\{R, S\}$, where R and S are binary relations. Let E be $R \cup \pi_{2,1}(S)$. Consider database D:

$$R(D) := \{(1, 2)\}$$
$$S(D) := \{(3, 4)\}$$

Let P be the singleton $\{(F, f)\}$, where F is an SA^{-rp} expression and f is a function from $\{1, 2\}$ to the set X, which can be either $\{1\}$ or $\{1, 2\}$. If X is $\{1\}$, then $f(1) = f(2) = 1$. It is clear that in this case each tuple (x, y) in the set $\{f(\overline{a}) \mid \overline{a} \in F(D)\}$ has $x = y$. If X is $\{1, 2\}$, then f can be the identical function or the permutation switching 1 and 2, i.e., $f(1) = 2$ and $f(2) = 1$. An easy inductive argument shows that each tuple (x, y) in the result of a binary SA^{-rp} expression F on database D will have $x < y$. Therefore, either each tuple (x, y) in the set $\{f(\overline{a}) \mid \overline{a} \in F(D)\}$ will have $x < y$ (if f is the identical function), or each tuple (x, y) in that set will have $x > y$ (if f permutes 1 and 2). In the set $E(D) = \{(1, 2), (4, 3)\}$, however, none of these three properties hold. □

Proof. The construction of the set P and the correctness proof are by structural induction. We will write P_E to denote that the set P corresponds to expression E.

1. If $E = R$, then $P_E := \{(R, \mathrm{Id})\}$, where Id denotes the identity function.

2. If $E = \sigma_\theta E_1$, then $P_E := \{(\sigma_{\theta_f} F, f) \mid (F, f) \in P_{E_1}\}$, where $\theta_f := \theta[x_i/x_{f(i)}]$. In proof:

$$\begin{aligned}
\overline{a} \in \sigma_\theta E_1(D) \quad &\Leftrightarrow \quad \overline{a} \in E_1(D) \text{ and } \overline{a} \text{ satisfies } \theta \\
&\Leftrightarrow \quad \overline{a} \in \bigcup_{(F,f)\in P_{E_1}} \{f(\overline{b}) \mid \overline{b} \in F(D)\} \quad \text{and} \quad \overline{a} \text{ satisfies } \theta \\
&\Leftrightarrow \quad \overline{a} \in \bigcup_{(F,f)\in P_{E_1}} \{f(\overline{b}) \mid \overline{b} \in \sigma_{\theta_f} F(D)\}
\end{aligned}$$

3. If $E = \pi_{i_1,\ldots,i_n} E_1$, then

$$P_E := \{(F, f'_{i_1,\ldots,i_n}) \mid (F, f) \in P_{E_1}\}$$

where $f'_{i_1,\ldots,i_n}$ is the function mapping j to $f(i_j)$ for all $1 \leqslant j \leqslant n$. In proof:

$$\begin{aligned}
&\overline{a} \in \pi_{i_1,\ldots,i_n} E_1(D)\\
&\quad\Leftrightarrow\quad \exists\, \overline{b} \in E_1(D) \text{ such that } (a_1,\ldots,a_n) = (b_{i_1},\ldots,b_{i_n})\\
&\quad\Leftrightarrow\quad \exists\, \overline{b} \in \bigcup_{(F,f)\in P_{E_1}} \{f(\overline{c}) \mid \overline{c} \in F(D)\}\\
&\qquad\qquad \text{such that } (a_1,\ldots,a_n) = (b_{i_1},\ldots,b_{i_n})\\
&\quad\Leftrightarrow\quad \overline{a} \in \bigcup_{(F,f)\in P_{E_1}} \{f'_{i_1,\ldots,i_n}(\overline{c}) \mid \overline{c} \in F(D)\}\\
&\qquad\qquad \text{where } f'_{i_1,\ldots,i_n} \text{ is defined as above.}
\end{aligned}$$

4. If $E = E_1 \cup E_2$, then $P_E := P_{E_1} \cup P_{E_2}$.

5. If $E = E_1 - E_2$, then

$$P_E := \{(F_1 - \bigcup_{(F_2,f_2)\in P_{E_2}} F_1 \ltimes_{\theta_{f_1\cap f_2}} F_2, f_1) \mid (F_1, f_1) \in P_{E_1}\}$$

where $\theta_{f_1\cap f_2} := \bigwedge_{i=1}^{n} x_{f_1(i)} = y_{f_2(i)}$. In proof:

$$\begin{aligned}
&\overline{a} \in E_1 - E_2(D)\\
&\quad\Leftrightarrow\quad \overline{a} \in \bigcup_{(F,f)\in P_{E_1}} \{f(\overline{b}) \mid \overline{b} \in F(D)\} - \bigcup_{(F,f)\in P_{E_2}} \{f(\overline{c}) \mid \overline{c} \in F(D)\}\\
&\quad\Leftrightarrow\quad \exists (F_1,f_1) \in P_{E_1}, \exists \overline{b} \in F_1(D) \;:\; \overline{a} = f_1(\overline{b})\\
&\qquad\qquad \text{and } \forall (F_2,f_2) \in P_{E_2}, \forall \overline{c} \in F_2(D) \;:\; \overline{a} \neq f_2(\overline{c})\\
&\quad\Leftrightarrow\quad \exists (F_1,f_1) \in P_{E_1}, \exists \overline{b} \in F_1(D) \;:\; \overline{a} = f_1(\overline{b})\\
&\qquad\qquad \text{and } \forall (F_2,f_2) \in P_{E_2}, \forall \overline{c} \in F_1 \ltimes_{\theta_{f_1\cap f_2}} F_2(D) \;:\; \overline{a} \neq f_1(\overline{c})\\
&\quad\Leftrightarrow\quad \exists (F_1,f_1) \in P_{E_1}, \exists \overline{b} \in F_1 - \bigcup_{(F_2,f_2)\in P_{E_2}} F_1 \ltimes_{\theta_{f_1\cap f_2}} F_2(D) \;:\; \overline{a} = f_1(\overline{b})\\
&\quad\Leftrightarrow\quad \overline{a} \in \bigcup_{(F_1,f_1)\in P_{E_1}} \big\{f_1(\overline{b}) \mid \overline{b} \in F_1 - \bigcup_{(F_2,f_2)\in P_{E_2}} F_1 \ltimes_{\theta_{f_1\cap f_2}} F_2(D)\big\}
\end{aligned}$$

6. If $E = E_1 \ltimes_\theta E_2$, then

$$P_E := \{(F_1 \ltimes_{\theta_{f_1 f_2}} F_2, f_1) \mid (F_1, f_1) \in P_{E_1}, (F_2, f_2) \in P_{E_2}\}$$

where $\theta_{f_1 f_2} := \theta[x_i/x_{f_1(i)}, y_j/y_{f_2(j)}]$. In proof:

$$
\begin{aligned}
&\overline{a} \in E_1 \ltimes_\theta E_2(D) \\
&\quad\Leftrightarrow\quad \overline{a} \in E_1(D) \text{ and } \exists \overline{b} \in E_2(D) \text{ such that } \theta(\overline{a}, \overline{b}) \text{ holds} \\
&\quad\Leftrightarrow\quad \overline{a} \in \bigcup_{(F_1,f_1)\in P_{E_1}} \{f_1(\overline{c}) \mid \overline{c} \in F_1(D)\} \\
&\qquad\qquad \text{and } \exists \overline{b} \in \bigcup_{(F_2,f_2)\in P_{E_2}} \{f_2(\overline{d}) \mid \overline{d} \in F_2(D)\} \text{ such that } \theta(\overline{a}, \overline{b}) \text{ holds} \\
&\quad\Leftrightarrow\quad \exists (F_1, f_1) \in P_{E_1}, \exists \overline{c} \in F_1(D) \ : \ \overline{a} = f_1(\overline{c}) \\
&\qquad\qquad \text{and } \exists (F_2, f_2) \in P_{E_2}, \exists \overline{d} \in F_2(D), \exists \overline{b} \ : \ \overline{b} = f_2(\overline{d}) \\
&\qquad\qquad \text{such that } \theta(f_1(\overline{c}), f_2(\overline{d})) \text{ holds} \\
&\quad\Leftrightarrow\quad \overline{a} \in \bigcup_{\substack{(F_1,f_1)\in P_{E_1} \\ (F_2,f_2)\in P_{E_2}}} \{f_1(\overline{c}) \mid \overline{c} \in F_1 \ltimes_{\theta_{f_1 f_2}} F_2(D)\}
\end{aligned}
$$

This concludes our proof. □

If one is only interested in "boolean" queries, i.e., yes/no properties of databases, which is often the case in practice, e.g., integrity constraints or decision queries, then we can strengthen our simulation result into a full equivalence result:

Corollary 7.7. *Let E be an SA expression of arity n. Then there exists an SA^{-rp} expression E' such that for each database D:*

$$E(D) \neq \varnothing \quad\Leftrightarrow\quad E'(D) \neq \varnothing.$$

Proof. From Theorem 7.4, it follows that $E(D) \neq \varnothing$ if and only if for some pair (F, f) in the set P_E, we have: $F(D) \neq \varnothing$. Note that each F is an SA^{-rp} expression. Expression E' is now defined as

$$\bigcup_{(F,f)\in P_E} \pi_{()} F.$$

where () is the empty projection list. □

7.1.3 Complexity issues

The algorithm in the proof of Theorem 7.4 has an exponential worst-case complexity. In order to make this statement precise, define size(E), for an SA expression E, as the number of operators in E. Furthermore, for a set P of pairs as in Theorem 7.4, define size(P) as the sum of the sizes of the SA^{-rp} expressions F in P, i.e., $\mathrm{size}(P) = \sum_{(F,f)\in P} \mathrm{size}(F)$. We then have:

Proposition 7.8. *Let E be an SA expression and let P_E be the set constructed by the algorithm in the proof of Theorem 7.4. Then,* $\mathrm{size}(P_E) \leqslant 2^{3\cdot \mathrm{size}(E)}$.

Proof. We simultaneously show by structural induction that $\text{size}(P_E) \leqslant 2^{3\cdot\text{size}(E)}$ and $|P_E| \leqslant 2^{\text{size}(E)}$. Here, we only present the case where $E = E_1 - E_2$. The other cases are similar but easier. For $E = E_1 - E_2$, we have

$$|P_E| = |P_{E_1}| \leqslant 2^{\text{size}(E_1)} \leqslant 2^{\text{size}(E_1)+\text{size}(E_2)+1} = 2^{\text{size}(E)},$$

and

$$\begin{aligned}
\text{size}(P_E) &= \text{size}(P_{E_1}) + |P_{E_1}| + \text{size}(P_{E_1}) \cdot |P_{E_2}| \\
&\quad + \text{size}(P_{E_2}) \cdot |P_{E_1}| + 2 \cdot |P_{E_1}| \cdot |P_{E_2}| \\
&\leqslant 2^{3\cdot\text{size}(E_1)} + 2^{\text{size}(E_1)} + 2^{3\cdot\text{size}(E_1)+\text{size}(E_2)} \\
&\quad + 2^{3\cdot\text{size}(E_2)+\text{size}(E_1)} + 2^{\text{size}(E_1)+\text{size}(E_2)+1} \\
&\leqslant 2^2 \cdot 2^{3\cdot\text{size}(E_1)+3\cdot\text{size}(E_2)} + 2^{\text{size}(E_1)+\text{size}(E_2)+1} \\
&\leqslant 2^{3\cdot\text{size}(E_1)+3\cdot\text{size}(E_2)+3} \\
&= 2^{3\cdot\text{size}(E)}
\end{aligned}$$

□

This upper bound is sharp. Indeed, let E be the expression

$$\Big(\big((\pi_{2,1}R - S_1) - S_2\big) - \cdots\Big) - S_n,$$

where R and S_i are binary relations for all i. Then the set P_E constructed by the algorithm in the proof of Theorem 7.4 is the singleton $\{(E_n, f)\}$, where $f(1) = 2$ and $f(2) = 1$, and where E_n is inductively defined as follows:

$$\begin{aligned}
E_1 &:= R - R \underset{\substack{2=1\\1=2}}{\ltimes} S_1 \\
E_{i+1} &:= E_i - E_i \underset{\substack{2=1\\1=2}}{\ltimes} S_{i+1} \quad (\text{for } 1 < i \leqslant n).
\end{aligned}$$

Clearly, the size of E_n is exponential in the size of E.

For this particular expression E, however, there is a set P'_E of pairs satisfying the conditions in Theorem 7.4 of size polynomial in the size of E. Indeed, note that expression E is equivalent to the expression $\pi_{2,1}F$, where F is

$$R - R \underset{\substack{2=1\\1=2}}{\ltimes} (S_1 \cup \cdots \cup S_n).$$

Therefore, a set P'_E polynomial in the size of E would be the singleton $\{(F, f)\}$.

The question whether, in Theorem 7.4 in general, such a polynomial-size set P always exists, remains open.

7.1.4 Conclusion

We have shown that SA expressions that employ repetitions and permutations in projection lists can be simulated using SA expressions that do not employ these features. The same holds for the full relational algebra RA, but there this is trivial to

prove, while here the proof is not trivial. There are probably no practical applications of the theorem, and indeed, neither are we aware of practical applications of the corresponding theorem for RA. Nevertheless, as already mentioned in the Introduction, the distinction between the "named" and the "unnamed" perspective in the relational model has received sufficient attention in a renowned textbook [1], and SA is a sufficiently important fragment of RA, so that it seems warranted, if only for the didactical purpose of thorough theoretical understanding of SA, to investigate the named–unnamed distinction for SA as well as for RA, as we have done in the present section.

7.2 An Ehrenfeucht-Fraïssé game for the semijoin algebra

In this section, we describe an Ehrenfeucht-Fraïssé game that characterizes the discerning power of the semijoin algebra. For technical reasons, we will restrict attention to the expressive power of SA^{-rp} expressions. This is not a limitation. Indeed, according to Theorem 7.4 and Corollary 7.7, SA and SA^{-rp} have equal expressive power. Furthermore, we assume that the set of selection predicates Ω_σ equals the set of semijoin predicates $\Omega_\ltimes$ (see Chapter 2), and we will refer to them by Ω.

Let A and B be two databases over the same schema $\mathbf{S}$. The *semijoin game* on these databases is played by two players, called the spoiler and the duplicator. They, in turn, choose tuples from the tuple spaces T_A and T_B, which are defined as follows:

$$T_A := \bigcup_{R\in\mathbf{S}} \bigcup_{k=1}^{\mathrm{arity}(R)} \big\{\pi_{i_1,\ldots,i_k}(A(R)) \mid i_1,\ldots,i_k \in \{1,\ldots,\mathrm{arity}(R)\} \text{ strictly increasing}\big\},$$

and T_B is defined analogously. So, the players can pick tuples from the databases and projections of these. The restriction to projections with a strictly increasing projection list, is only for technical reasons, which will become clear in the proof of Theorem 7.11.

At each stage in the game, there is a tuple $\overline{a} \in T_A$ and a tuple $\overline{b} \in T_B$. We will denote such a configuration by $(A, \overline{a}; B, \overline{b})$. The conditions for the duplicator to win the game with 0 rounds are:

1. $\forall R \in \mathbf{S}$, $\forall i_1,\ldots,i_k \in \{1,\ldots,\mathrm{arity}(R)\}$ strictly increasing:

   $$\overline{a} \in \pi_{i_1,\ldots,i_k}(A(R)) \Leftrightarrow \overline{b} \in \pi_{i_1,\ldots,i_k}(B(R)),$$

   where k is the arity of $\overline{a}$ and $\overline{b}$

2. for every atomic formula (equivalently, for every quantifier-free formula) θ over Ω, $\theta(\overline{a})$ holds iff $\theta(\overline{b})$ holds.

In the game with $m \geqslant 1$ rounds, the spoiler will be the first one to make a move. Therefore, he first chooses a database (A or B). Then he picks a tuple in T_A or in T_B respectively. The duplicator then has to make an "analogous" move in the other tuple

space. When the duplicator can hold this for m times, no matter what moves the spoiler takes, we say that the duplicator wins the m-round semijoin game on A and B. The "analogous" moves for the duplicator are formally defined as legal answers in the next definition.

Definition 7.9 (legal answer)**.** Suppose that at a certain moment in the semijoin game, the configuration is $(A, \overline{a}; B, \overline{b})$. If the spoiler takes a tuple $\overline{c} \in T_A$ in his next move, then the tuples $\overline{d} \in T_B$, for which the following conditions hold, are legal answers for the duplicator:

1. $\forall R \in \mathbf{S}$, $\forall i_1, \ldots, i_k \in \{1, \ldots, \text{arity}(R)\}$ strictly increasing:

$$\overline{c} \in \pi_{i_1,\ldots,i_k}(A(R)) \Leftrightarrow \overline{d} \in \pi_{i_1,\ldots,i_k}(B(R)),$$

   where k is the arity of $\overline{c}$ and $\overline{d}$

2. for every atomic formula θ over Ω, $\theta(\overline{a}, \overline{c})$ holds iff $\theta(\overline{b}, \overline{d})$ holds.

If the spoiler takes a tuple $\overline{d} \in T_B$, the legal answers $\overline{c} \in T_A$ are defined identically.

In the following, we denote the semijoin game with initial configuration $(A, \overline{a}; B, \overline{b})$ and that consists of m rounds, by $G_m(A, \overline{a}; B, \overline{b})$.

We first state and prove

Proposition 7.10. *If the duplicator wins $G_m(A, \overline{a}; B, \overline{b})$, then for each SA^{-rp} expression E with $\leqslant m$ nested semijoins and projections, we have $\overline{a} \in E(A) \Leftrightarrow \overline{b} \in E(B)$.*

Proof. We prove this by induction on m. The base case $m = 0$ is clear. Now consider the case $m > 0$. Suppose that $\overline{a} \in E_1 \ltimes_\theta E_2(A)$ but $\overline{b} \notin E_1 \ltimes_\theta E_2(B)$. Then $\overline{a} \in E_1(A)$ and $\exists \overline{c} \in E_2(A) : \theta(\overline{a}, \overline{c})$, and either (*) $\overline{b} \notin E_1(B)$ or (**) $\neg \exists \overline{d} \in E_2(B) : \theta(\overline{b}, \overline{d})$. In situation (*), $\overline{a}$ and $\overline{b}$ are distinguished by an expression with $m - 1$ semijoins or projections, so the spoiler has a winning strategy; in situation (**), the spoiler has a winning strategy by choosing this $\overline{c} \in E_2(A)$ with $\theta(\overline{a}, \overline{c})$, because each legal answer of the duplicator $\overline{d}$ has $\theta(\overline{b}, \overline{d})$ and therefore $\overline{d} \notin E_2(B)$. So, the spoiler now has a winning strategy in the game $G_{m-1}(A, \overline{c}; B, \overline{d})$. In case a projection distinguishes $\overline{a}$ and $\overline{b}$, a similar winning strategy for the spoiler exists. In case $\overline{a}$ and $\overline{b}$ are distinguished by an expression that is neither a semijoin, nor a projection, there is a simpler expression that distinguishes them, so the result follows by structural induction. □

We now come to the main theorem concerning the discerning power of SA. This theorem concerns the game $G_\infty(A, \overline{a}; B, \overline{b})$, which we also abbreviate as $G(A, \overline{a}; B, \overline{b})$. We say that the duplicator wins $G(A, \overline{a}; B, \overline{b})$ if the spoiler has no winning strategy. This means that the duplicator can keep on playing forever, choosing legal answers for every move of the spoiler.

Theorem 7.11. *The duplicator wins $G(A, \overline{a}; B, \overline{b})$ if and only if for each SA^{-rp} expression E, we have $\overline{a} \in E(A) \Leftrightarrow \overline{b} \in E(B)$.*

Proof. The 'only if' direction of the proof follows directly from Proposition 7.10, because if the duplicator wins $G(A, \overline{a}; B, \overline{b})$, he wins $G_m(A, \overline{a}; B, \overline{b})$ for every $m \geqslant 0$. So,

Table 7.1: Queries delineating the expressive power of $\mathrm{SA}^{\neq}$.

Expressible	Inexpressible
$R \times S \ \cap \ T$ $T \subseteq R \times S$	$R \times S \subseteq T$ $T = R \times S$ $R \circ S \ \cap \ T$ $T \subseteq R \circ S$ $R \circ S \subseteq T$
$\exists$ path of length k $\exists$ simple path of length k $(k \leqslant 2)$	$\exists$ simple path of length k $(k \geqslant 3)$
$\exists$ cycle of length k $(k \leqslant 2)$	$\exists$ cycle of length k $(k \geqslant 3)$
	$\exists \geqslant k$ elements $(k \geqslant 3)$

$\overline{a}$ and $\overline{b}$ are indistinguishable through all semijoin expressions. For the 'if' direction, it is sufficient to prove that if the duplicator loses, $\overline{a}$ and $\overline{b}$ are distinguishable. We therefore construct, by induction, an SA^{-rp} expression $E^m_{\overline{a}}$ such that (i) $\overline{a} \in E^m_{\overline{a}}(A)$, and (ii) $\overline{b} \in E^m_{\overline{a}}(B)$ iff the duplicator wins $G_m(A, \overline{a}; B, \overline{b})$. We define $E^0_{\overline{a}}$ as

$$\sigma_{\theta_{\overline{a}}}\Big(\bigcap_{R\in\mathbf{S}} \bigcap_{\{\overline{L}\in Z^k \mid \overline{a}\in\pi_{\overline{L}}(A(R))\}} \pi_{\overline{L}}(R)\Big) - \bigcup_{R\in\mathbf{S}} \bigcup_{\{\overline{L}\in Z^k \mid \overline{a}\notin\pi_{\overline{L}}(A(R))\}} \pi_{\overline{L}}(R)$$

In this expression, Z is a shorthand for $\{1, \ldots, \mathrm{arity}(R)\}$; k is the arity of $\overline{a}$; and $\theta_{\overline{a}}$ is the *atomic type* of $\overline{a}$ over Ω, i.e., the conjunction of all atomic and negated atomic formulas over Ω that are true of $\overline{a}$. Furthermore, the projection list $\overline{L}$ is supposed to be strictly increasing.

We now construct $E^m_{\overline{a}}$ in terms of $E^{m-1}_{\overline{a}}$:

$$\bigcap_{\overline{c}\in T_A} \big(E^0_{\overline{a}} \ltimes_{\theta_{\overline{a},\overline{c}}} E^{r-1}_{\overline{c}}\big) \cap \Big(E^0_{\overline{a}} - \bigcup_{j=1}^{s} \bigcup_{\theta} \big(E^0_{\overline{a}} \ltimes_{\theta} \bigcap_{\substack{\overline{c}\in T_A \\ \theta(\overline{a},\overline{c})}} (E^{m-1}_{\overline{c}})^{\mathrm{compl}}\big)\Big)$$

In this expression, $\theta_{\overline{a},\overline{c}}$ is the atomic type of $\overline{a}$ and $\overline{c}$ over Ω; s is the maximal arity of a relation in $\mathbf{S}$; θ ranges over all atomic Ω-types of two tuples, one with the arity of $\overline{a}$, and one with arity j. The notation E^{compl}, for an expression of arity k, is a shorthand for

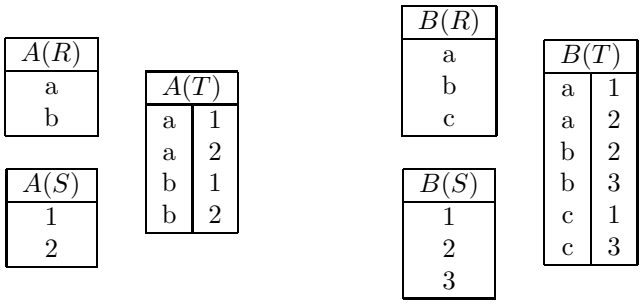
$$\Big(\bigcup_{R\in\mathbf{S}} \bigcup_{\overline{L}\in\{1,\ldots,\mathrm{arity}(R)\}^k} \pi_{\overline{L}}(R)\Big) - E$$

□

7.2.1 The expressive power of $\mathrm{SA}^{\neq}$

In this section, we present some queries that delineate the expressive power of $\mathrm{SA}^{\neq}$. They are summarized in Table 7.1. The operation $R \circ S$ for binary relations R and S is a shorthand for $\pi_{1,4}\big(\sigma_{2=3}(R \times S)\big)$.

We now discuss the results presented in the table. The semijoin algebra lacks the Cartesian product operator, but nevertheless one can check if $T \subseteq R \times S$. Indeed,



$A(R)$
a
b

$A(S)$
1
2

$A(T)$	
a	1
a	2
b	1
b	2

$B(R)$
a
b
c

$B(S)$
1
2
3

$B(T)$	
a	1
a	2
b	2
b	3
c	1
c	3

Figure 7.1: In A, $T = R \times S$, but not in B.

$A(R)$	
1	a
3	b

$A(S)$	
a	2
b	4

$A(T)$	
1	2
3	4

$B(R)$	
1	a
3	b

$B(S)$	
b	2
a	4

$B(T)$	
1	2
3	4

Figure 7.2: In A, $T = R \circ S$, but in B neither $T \subseteq R \circ S$ nor $T \supseteq R \circ S$.

$T \subseteq R \times S$ iff $T - (T \cap R \times S) = \varnothing$, and $T \cap R \times S = (T \ltimes_{1=1 \wedge 2=2} R) \ltimes_{3=1 \wedge 4=2} S$. Conversely, it is impossible to check if $T \supseteq R \times S$. In Figure 7.1, two databases A and B are shown that are indistinguishable through semijoin expressions because the duplicator has an obvious winning strategy. But A satisfies $T \supseteq R \times S$ and B does not. The same databases actually show that it is impossible to check if $T = R \times S$.

Although one can check in SA if a relation is contained in a *Cartesian product*, it is impossible to check if a relation is contained in or subsumed by a *join*. Using our semijoin game, one can show that databases A and B in Figure 7.2 satisfy the same semijoin expressions. But A satisfies $T = R \circ S$, while B satisfies neither $T \subseteq R \circ S$ nor $T \supseteq R \circ S$. Note that a binary relation R is transitive if and only if $R \circ R \subseteq R$. This is a special case of $R \circ S \subseteq T$; yet, a similar argument shows that transitivity is also inexpressible in the semijoin algebra.

The existence of a path of length k can be checked with the following inductively defined semijoin expression:

$$\begin{cases} \text{path}(1) & := & R \\ \text{path}(k) & := & R \ltimes_{2=1} \big(\text{path}(k-1)\big) \end{cases}$$

Problems arise when we require the path to be simple. Let $D^{(k)}$ be the structure $\{(1,2),(2,3),\ldots,(k-1,k),(k,1)\}$ over the schema $\mathbf{S}$ containing a single edge relation R. Then, the duplicator has a winning strategy in the infinite game played on $D^{(k)}$ and $D^{(k+1)}$ where $k \geqslant 4$. To see this, note that only three types of moves are possible here: next tuple (change only first component of pebbled tuple), previous tuple (change only second component) and other tuple (change both components). The duplicator can answer every type of move of the spoiler. But $D^{(k+1)}$ contains a simple path of length k and $D^{(k)}$ does not. For $k = 3$, note that $D^{(3)}$ and $D^{(4)}$ *are* distinguishable. Nevertheless, existence of a simple path of length 3 is still inexpressible because $D^{(4)}$ is indistinguishable from the structure consisting of two disjoint copies of $D^{(3)}$. For $k = 2$, the existence of a path of length 2 is expressible as $R \ltimes_{x_2=y_1 \wedge x_2 \neq x_1 \wedge y_2 \neq x_2} R$.

Another property that is inexpressible in $\mathrm{SA}^{\neq}$ is the existence of a cycle of length k. For $k \geqslant 4$, the inexpressibility result follows because $D^{(k)}$ contains a cycle of length k and $D^{(k+1)}$ does not. For $k = 3$, that the structure consisting of two disjoint copies of $D^{(3)}$ contains a cycle of length 3, but $D^{(4)}$ does not.

A last example of a query that is inexpressible in $\mathrm{SA}^{\neq}$ is the query that asks if there are at least k elements in a unary relation S, where $k \geqslant 3$. This property is inexpressible because the duplicator has a winning strategy in the infinite game played on two relations, one with 2 and one with k distinct elements.

7.2.2 Impact of order: the expressive power of $\mathrm{SA}^{<,<}$

In this section, we investigate the impact of order. On ordered databases (where Ω now also contains a total order on the domain), the query that asks if there are at least k elements in a unary relation S becomes expressible as $\mathrm{at\_least}(k)$, which is inductively defined as follows:

$$\begin{cases} \mathrm{at\_least}(1) & := & S \\ \mathrm{at\_least}(k) & := & S \ltimes_{1<1} \big(\mathrm{at\_least}(k-1)\big) \end{cases}$$

Note that this query is independent of the order. This is very interesting because in first-order logic, there also exists an order-invariant query that is expressible with but inexpressible without order ([1, Exercise 17.27] and [17, Proposition 2.5.6]).

Some inexpressible queries presented in Section 7.2.1 remain inexpressible on ordered databases. An example is the query $R \times S \subseteq T$. Indeed, consider the following databases A and B: $A(R) = B(R) = \{1, 2, \ldots, m\}$, $A(S) = B(S) = \{m+1, m+2, \ldots, 2m\}$, $A(T) = A(R) \times A(S)$ and $B(T) = A(T) - \{(\frac{m+1}{2}, m + \frac{m+1}{2})\}$. We will show that when $m = 2n + 1$, the duplicator has a winning strategy in the n-round semijoin game $G_n(A, \langle\rangle; B, \langle\rangle)$ with $\Omega = \{=, <\}$. From Lemma 7.10, it then follows that the query $R \times S \subseteq T$ is inexpressible in $\mathrm{SA}^{<,<}$. The duplicator's winning strategy consists of playing exact match until the spoiler chooses $\overline{c}$ to be the special tuple $(\frac{m+1}{2}, m + \frac{m+1}{2})$ in A. In that case we must distinguish five possibilities for the previous tuple $\overline{a}$: (1) $a_1 = \frac{m+3}{2}$, (2) $a_1 = \frac{m-1}{2}$, (3) $a_1 = \frac{m+1}{2}$ and $a_2 = m + \frac{m+3}{2}$, (4) $a_1 = \frac{m+1}{2}$ and $a_2 = m + \frac{m-1}{2}$ and (5) all other cases. The duplicator chooses $\overline{d}$ equal to $(\frac{m-1}{2}, m + \frac{m+1}{2})$ in case 1, $(\frac{m+3}{2}, m + \frac{m+1}{2})$ in case 2, $(\frac{m+1}{2}, m + \frac{m-1}{2})$ in case 3, $(\frac{m+1}{2}, m + \frac{m+3}{2})$ in case 4, and $(\frac{m-1}{2}, m + \frac{m+1}{2})$ in case 5. Let us assume case 1 applies; cases 2 to 5 are analogous. Then, there are two possibilities. First, if the spoiler chooses a value $c_1 \neq a_1 - 1$ or if he chooses a value $d_1 \neq b_1 + 1$ in some next round, the duplicator can play exact match and the game starts over. Second, if the spoiler chooses in each next round $c_1 = a_1 - 1$ or $d_1 = b_1 + 1$, the duplicator answers $d_1 = b_1 - 1$ or $c_1 = a_1 + 1$, respectively. The duplicator can follow this strategy for at least $\frac{m-3}{2} = n - 1$ rounds. Counting from the round where the spoiler chose the special tuple, we thus see that the duplicator wins the game $G_n(A, \langle\rangle; B, \langle\rangle)$.

Exactly the same argument shows that also the query $R \times S = T$ is inexpressible in SA with order.

Another query from Table 7.1 that remains inexpressible in SA with order is $R \circ S \subseteq T$. Therefore, consider the following databases A and B: $A(R) = B(R) = \{1, \ldots, m\} \times \{2m+1\}$, $A(S) = B(S) = \{2m+1\} \times \{m+1, \ldots, 2m\}$, $A(T) = A(R) \circ$

$A(S) = \{1, \ldots, m\} \times \{m+1, \ldots, 2m\}$ and $B(T) = B(R) \circ B(S) - \{(\frac{m+1}{2}, m + \frac{m+1}{2})\}$. A similar argument as in the previous paragraph shows that when $m = 2n + 1$, the duplicator wins $G_n(A, \langle\rangle; B, \langle\rangle)$. Again, this also shows that $R \circ S = T$ is inexpressible in SA with order.

For the remaining SA-inexpressible queries in Table 7.1, the question whether they become expressible in SA with order remains open.

Publications

The results presented in this thesis have been published in several papers with several different coauthors. We survey these publications here. (The references refer to the Bibliography.)

Chapter	Reference
3	[44]
4	[42, 43]
5	[29, 30]
6	[33]
7.1	submitted
7.2	[45]

Bibliography

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

[2] A. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[3] H. Andreka, I. Hodkinson, and I. Németi. Finite algebras of relations are representable on finite sets. *Journal of Symbolic Logic*, 64(1):243–267, 1999.

[4] H. Andréka, I. Németi, and J. van Benthem. Modal languages and bounded fragments of predicate logic. *Journal of Philosophical Logic*, 27(3):217–274, 1998.

[5] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems*, 29(1):162–194, March 2004.

[6] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, June 2006.

[7] A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*, 33(3):6–12, 2004.

[8] F. Ayres and E. Mendelson. *Schaum's Outline of Calculus*. McGraw-Hill, 1999.

[9] 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.

[10] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E.F. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S.B. Zdonik. Retrospective on aurora. *The VLDB Journal*, 13(4):370–383, 2004.

[11] C. Beeri, R. Fagin, D. Maier, and M. Yannakakis. On the desirability of acyclic database schemes. *Journal of the ACM*, 30(3):479–513, 1983.

[12] P.A. Bernstein and D.W. Chiu. Using semi-joins to solve relational queries. *Journal of the ACM*, 28(1):25–40, 1981.

[13] P.A. Bernstein and N. Goodman. Power of natural semijoins. *SIAM Journal on Computing*, 10(4):751–771, 1981.

[14] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[15] E.F. Codd. Relational completeness of data base sublanguages. In R. Rustin, editor, *Data Base Systems*, pages 65–98. Prentice-Hall, 1972.

[16] H. de Nivelle and M. de Rijke. Deciding the guarded fragments by resolution. *Journal of Symbolic Computation*, 35(1):21–58, 2003.

[17] H.-D. Ebbinghaus and J. Flum. *Finite Model Theory*. Springer, 1999.

[18] J. Flum, M. Frick, and M. Grohe. Query evaluation via tree-decompositions. *Journal of the ACM*, 49(6):716–752, 2002.

[19] H. Garcia-Molina, J.D. Ullman, and J. Widom. *Database Systems: the Complete Book*. Prentice Hall, 2000.

[20] L. Golab and M.T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases*, 2003.

[21] G. Gottlob, E. Grädel, and H. Veith. Datalog lite: a deductive query language with linear time model checking. *ACM Transactions on Computational Logic*, 3(1):42–79, 2002.

[22] E. Grädel. Decision procedures for guarded logics. In *Proceedings of the 16th International Conference on Automated Deduction*, volume 1632. Springer-Verlag, 1999.

[23] E. Grädel. On the restraining power of guards. *Journal of Symbolic Logic*, 64(4):1719–1742, 1999.

[24] E. Grädel. Guarded fixed point logics and the monadic theory of countable trees. *Theoretical Computer Science*, 288(1):129–152, 2002.

[25] E. Grädel, C. Hirsch, and M. Otto. Back and forth between guarded and modal logics. *ACM Transactions on Computational Logic*, 3(3):418–463, 2002.

[26] E. Grädel and I. Walukiewicz. Guarded fixed point logic. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science LICS '99*, pages 45–54, 1999.

[27] G. Graefe. Relational division: four algorithms and their performance. In *Proceedings of the 5th International Conference on Data Engineering*, pages 94–101. IEEE Computer Society, 1989.

[28] G. Graefe and R.L. Cole. Fast algorithms for universal quantification in large databases. *ACM Transactions on Database Systems*, 20(2):187–236, 1995.

[29] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. Van den Bussche. Database query processing using finite cursor machines. In *Proceedings of the 11th International Conference on Database Theory*, volume 4353 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 2007.

[30] M. Grohe, Y. Gurevich, D. Leinders, N. Schweikardt, J. Tyszkiewicz, and J. Van den Bussche. Database query processing using finite cursor machines. *Theory of Computing Systems*, To appear. Special Issue with selected papers from ICDT 2007.

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

[32] Y. Gurevich. Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1(1):77–111, July 2000.

[33] Y. Gurevich, D. Leinders, and J. Van den Bussche. A theory of stream queries. In M. Arenas and M.I. Schwartzbach, editors, *Proceedings of the 11th International Symposium on Database Programming Languages*, volume 4797 of *Lecture Notes in Computer Science*, pages 153–168. Springer Berlin / Heidelberg, September 2007.

[34] P. Halmos. *Algebraic Logic*. American Mathematical Society, 1962.

[35] J.Y. Halpern. The effect of bounding the number of primitive propositions and the depth of nesting on the complexity of modal logic. *Artificial Intelligence*, 75(2):361–372, 1995.

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

[37] S. Helmer and G. Moerkotte. Evaluation of main memory join algorithms for joins with set comparison join predicates. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 386–395. Morgan Kaufmann Publishers Inc., 1997.

[38] J.G. Hocking and G.S. Young. *Topology*. Dover Publications, 1988.

[39] E. Hoogland and M. Marx. Interpolation and definability in guarded fragments. *Studia Logica*, 70(3):373–409, April 2002.

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

[41] 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.

[42] 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.

[43] D. Leinders and J. Van den Bussche. On the complexity of division and set joins in the relational algebra. *Journal of Computer and System Sciences*, 73(4):538–549, June 2007. Special Issue with selected papers on database theory.

[44] 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, June 2005.

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

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

[47] S.R. Madden, M.J. Franklin, J.M Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1):122–173, March 2005.

[48] N. Mamoulis. Efficient processing of joins on set-valued attributes. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–168. ACM Press, 2003.

[49] M. Marx. Tolerance logic. *Journal of Logic, Language and Information*, 10(3):353–374, 2001.

[50] M. Marx and Y. Venema. Local variations on a loose theme: modal logic and decidability. In E. Grädel, P. Kolaitis, L. Libkin, M. Marx, J. Spencer, M. Vardi, Y. Venema, and S. Weinstein, editors, *Finite Model Theory and its Applications*. Springer-Verlag, 2004.

[51] I. Németi. A fine-structure analysis of first-order logic. In M. Marx, L. Pólos, and M. Masuch, editors, *Arrow Logic and Multimodal Logics*, Studies in Logic, Language and Information, pages 221–247. CSLI Publications, Stanford, 1995.

[52] K. Ramasamy, J.M. Patel, J.F. Naughton, and R. Kaushik. Set containment joins: the good, the bad and the ugly. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 351–362. Morgan Kaufmann Publishers Inc., 2000.

[53] S.G. Rao, A. Badia, and D. Van Gucht. Providing better support for a class of decision support queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 217–227. ACM Press, 1996.

[54] 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.

[55] S. Sarawagi and A. Kirpal. Efficient set joins on similarity predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 743–754. ACM Press, 2004.

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

[57] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1992.

[58] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[59] J. van Benthem. Dynamic bits and pieces. Technical Report LP-1997-01, ILLC, 1997.

[60] K. Weihrauch. *Computable Analysis: an introduction*. Springer-Verlag, 2000.

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

Samenvatting

In 1970 stelde Codd het intussen zeer bekende relationeel model voor. In het relationeel model wordt een gegevensbank voorgesteld als een eindige verzameling relaties. Een relatie is op haar beurt een eindige verzameling van tupels. Om queries, dit zijn vragen over de gegevens, uit te drukken in het relationeel model stelde Codd de relationele algebra (RA) voor met operatoren selectie, projectie, unie, verschil en join [14]. Sindsdien is de relationele algebra erg grondig bestudeerd [1]. Een zeer belangrijk resultaat is dat de uitdrukkingskracht van de relationele algebra gelijk is aan die van eerste-orde logica [15].

De "semijoin"-operator selecteert de tupels in de ene relatie die zouden deelnemen aan de volledige join met een andere relatie. De operator is dus uitdrukbaar met de andere operatoren in de relationele algebra, meer bepaald door gebruik te maken van de join- en de projectie-operator. Deze operator is ook grondig bestudeerd in het verleden. Bijvoorbeeld, terwijl het berekenen van een project-join query NP-compleet is in de grootte van de query en de gegevensbank, kan deze berekening in polynomiale tijd gebeuren als het schema van de gegevensbank acyclisch is [61]. En deze acycliciteit komt precies overeen met het bestaan van een programma van semijoins [11, 13, 12]. Semijoins worden vaak gebruikt om bij het verwerken van een query de gegevensbank voor te bereiden. Op die manier kunnen tupels die niet joinen worden geëlimineerd en de gegevensbank herleid worden tot het gedeelte dat effectief nodig is om het antwoord op de query te berekenen. Een andere interessante eigenschap is dat het resultaat van de semijoin-operatie lineair is in de grootte van de invoer relaties. Daarom zal de query processor de join-operatie overal waar mogelijk vervangen door de semijoin-operatie. Deze techniek staat bekend als "pushing projections" [19]. Wanneer de gegevensbank fysisch verspreid is over verschillende computers kan deze techniek veel netwerkverkeer helpen besparen en zo een aanzienlijke tijdwinst opleveren.

Merkwaardig genoeg echter is — voor zover we weten — de "semijoin algebra", dit is de algebra die we bekomen door de join-operatie in de relationele algebra te vervangen door de semijoin-operatie, nog nooit bestudeerd.

We tonen aan dat de semijoin algebra (SA) dezelfde uitdrukkingskracht heeft als het guarded fragment van eerste-orde logica. Dit fragment is geïntroduceerd door Andréka, van Benthem en Németi [4] met als doel modale logica uit te breiden van zogenoemde Kripke-structuren naar willekeurige relationele structuren zonder daarbij de interessante eigenschappen, zoals de "finite model property", te moeten opgeven.

Ook het guarded fragment is erg grondig bestudeerd [26, 23, 25, 24, 39].

Deze "Codd-stelling" voor de semijoin algebra heeft een aantal interessante gevolgen. Een eerste gevolg is dat de semijoin algebra de interessante eigenschappen van het guarded fragment overerft. Eén van de belangrijkste eigenschappen voor "database query processing" is wel beslisbaarheid. Concreet is het volgende probleem, dat het "*satisfiability* probleem" wordt genoemd, beslisbaar:

Invoer: Een SA expressie E.

Uitvoer: Bestaat er een gegevensbank D zodat het resultaat van de evaluatie van E op D niet leeg is?

Een onmiddellijk gevolg is het equivalentie-probleem voor de semijoin algebra ook beslisbaar is. Er is dus een algoritme dat controleert of twee gegeven SA expressies altijd hetzelfde resultaat zullen geven, wat ook de gegevensbank is waarop de expressies worden geëvalueerd. Dit suggereert meteen dat er algoritmes zijn om SA expressies te herschrijven in equivalente SA expressies die efficiënter kunnen worden geëvalueerd. Merkwaardig genoeg is het satisfiability probleem niet beslisbaar voor eerste-orde logica en dus ook niet voor de relationele algebra en voor SQL zonder groepering en aggregatie.

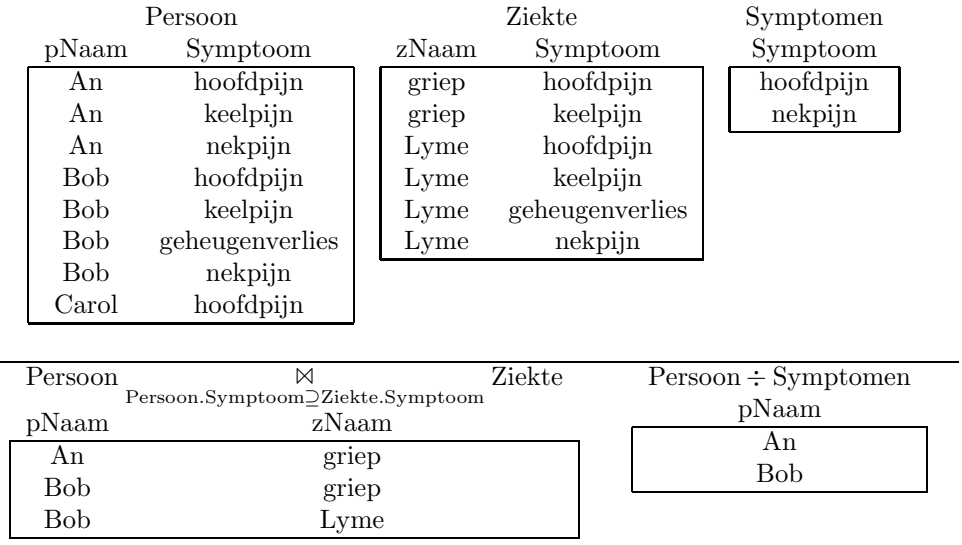
Een ander gevolg, gerelateerd aan het vorige, is dat we de tijdscomplexiteit van het satisfiability probleem voor SA hebben kunnen vastleggen door gebruik te maken van de tijdscomplexiteit van datzelfde probleem voor het guarded fragment. Het satisfiability probleem voor SA is EXPTIME-compleet.

De Codd-stelling voor SA heeft ook een aantal toepassingen. Een eerste toepassing is aantonen dat een bepaalde query niet kan worden uitgedrukt in SA. Om aan te tonen dat een query niet uitdrukbaar is in het guarded fragment, bestaat er immers al een techniek. Deze techniek is bekend als "guarded bisimulation". Andréka, van Benthem en Németi hebben aangetoond dat de verzameling eerste-orde formules die invariant zijn onder guarded bisimulations precies overeenkomt met de verzameling formules uit het guarded fragment [4].

In Hoofdstuk 2 definiëren we de nodige begrippen uit de logica en uit de theorie van gegevensbanken. De Codd-stelling, haar gevolgen en de bovenvermelde toepassing worden uiteengezet in Hoofdstuk 3 van deze tekst.

Een tweede toepassing van de Codd-stelling voor SA bespreken we in Hoofdstuk 4. Het betreft het evalueren van queries in lineaire ruimte. Beschouw een RA expressie E als lineair als voor elke gegevensbank D en voor elke deelexpressie E' van E, de grootte van het resultaat van de evaluatie van E' op invoer D lineair is in de grootte van D. Met andere woorden, bij de evaluatie van een lineaire RA expressie zal elk tussenresultaat een lineaire grootte hebben in verhouding tot de grootte van de invoer-gegevensbank. We tonen aan dat een query uitdrukbaar is door een lineaire RA expressie als en slechts als ze uitdrukbaar is door een SA expressie. We zullen tegelijkertijd aantonen dat een RA expressie ofwel lineair is, ofwel kwadratisch.

Dit resultaat kan verklaren waarom bepaalde operaties zwaar zijn voor de query processor. We gaan in deze tekst dieper in op de delingsoperatie en op de meer algemene "set joins". De delingsoperatie voor relaties werd al door Codd geïntroduceerd [15] en is hét typevoorbeeld van een set join. Set joins relateren elementen in de gegevensbank op basis van verzamelingen van waarden in plaats van op basis van één enkele



Figuur 1: Illustratie van de delingsoperatie en set-containment join.

waarde, zoals het geval is bij een standaard join-operatie. Concreet is het resultaat van de deling van $R(A, B)$ door $S(B)$ de verzameling van A-waarden waarvoor de verzameling B-waarden die door R met A gerelateerd zijn, de hele verzameling S omvat. Er bestaat ook een variant van deze delingsoperatie waarbij de verzameling gerelateerde B-waarden gelijk moet zijn aan de verzameling S. Algemener nog hebben we de "set-containment join" $R \underset{B \supseteq D}{\bowtie} S$ van $R(A, B)$ en $S(C, D)$, die als resultaat de verzameling

$$\big\{(a, c) \mid \{b \mid R(a, b)\} \supseteq \{d \mid S(c, d)\}\big\},$$

geeft, en opnieuw de analoge "set-equality join". In principe kan elk op verzamelingen gedefinieerd predikaat worden gebruikt in de plaats van $\supseteq$ en $=$ [53, 55]. Merk op dat de set join met als predikaat "de doorsnede is niet leeg" neerkomt op een gewone equijoin!

Voorbeeld 1. Figuur 1 illustreert de delingsoperatie en de set containment join. Het bovenste gedeelte toont drie relaties Persoon, Ziekte en Symptomen. Persoon relateert personen en symptomen; Ziekte relateert ziektes en symptomen; en Symptomen is een verzameling symptomen. De set containment join van Persoon en Ziekte op Persoon.Symptoom $\supseteq$ Ziekte.Symptoom is te zien links in het onderste gedeelte. De join legt het verband tussen een persoon en een ziekte als die persoon alle symptomen heeft die met de ziekte gepaard gaan. De deling van Persoon en Symptomen is te zien rechts beneden en toont de personen die alle symptomen hebben uit de verzameling Symptomen. □

Men heeft in het verleden vaak moeten vaststellen dat het berekenen van delingen met klassieke query processors zeer tijdrovend was [27, 28]. Set joins zijn inderdaad wel uitdrukbaar in de relationele algebra door gebruik te maken van equijoins en van de verschil-operatie, maar de expressies hiervoor blijken altijd vrij ingewikkeld en

inefficiënt. We tonen in deze tekst aan dat de deling en (testen op leegheid van) de set containment/equality join niet kunnen worden uitgedrukt in de semijoin algebra. Daarom moet elke RA expressie voor deze operaties tussenresultaten van kwadratische grootte geven. Op deze manier rechtvaardigen we dus eigenlijk formeel al het werk dat door vele auteurs is verricht om set joins te implementeren als "special-purpose" operatoren, of om ze te vertalen naar de krachtigere algebra met groepering, sortering en aggregratie [37, 48, 52]. Zo kunnen de deling en set-equality join efficiënt worden geïmplementeerd in $O(n \log n)$ tijd door trucs te gebruiken gebaseerd op sorteren en tellen.

Het berekenen van het resultaat van een semijoin algebra expressie kan dus gebeuren in lineaire ruimte. Maar hoe vaak moeten we nu de gegevens uit de gegevensbank lezen om tot het resultaat van een SA expressie te komen? We beantwoorden deze vraag in Hoofdstuk 5. In database query processing wordt een belangrijk onderscheid gemaakt tussen 1-pass en 2-pass algoritmen [19]. 1-pass algoritmes lezen de gegevens slechts één keer van de harde schijf. 2-pass algoritmes lezen de gegevens een eerste keer van de harde schijf, verwerken deze gegevens, schrijven de gegevens naar de harde schijf en lezen dan de gegevens opnieuw van de harde schijf om het resultaat te kunnen berekenen. Het verwerken van de gegevens nadat deze een eerste keer van de harde schijf zijn gelezen komt meestal neer op het sorteren ervan. Het is onmiddellijk duidelijk dat de selectie-operatie kan geïmplementeerd worden door een 1-pass algoritme. Elk tupel van de gegevensbank kan immers onafhankelijk van de andere tupels worden verwerkt. Als we dubbels even buiten beschouwing laten, dan kunnen ook de projectie- en de unie-operatie geïmplementeerd worden door een 1-pass algoritme. De verschil- en de semijoin-operatie echter kunnen intuïtief niet worden geïmplementeerd door een 1-pass algoritme.

Om dit nu echt hard te maken voeren we een model in dat we finite cursor machines (FCMs) noemen. Een FCM werkt op een aantal lijsten van tupels en kan zich op elk moment in een bepaalde toestand bevinden. Er zijn slechts een eindig aantal beschikbare toestanden. Een FCM heeft ook een intern geheugen bestaande uit een eindig aantal registers waarin bit strings kunnen worden opgeslaan. Het lezen van de tupels uit de lijsten gebeurt door middel van een eindig aantal cursors. Elke cursor kan op ieder ogenblik slechts één tupel lezen. Als resultaat kan een lijst van tupels worden geproduceerd. Om een realistisch model te zijn voor het sequentieel verwerken van de tupels in een gegevensbank, worden er een aantal beperkingen opgelegd: ten eerste kunnen de cursors maar in één richting over de lijsten lopen, tupel per tupel. Eens de laatste cursor een tupel van een lijst heeft verlaten, kan dat tupel dus nooit meer worden gelezen tijdens de berekening. Ten tweede is het intern geheugen van een FCM gelimiteerd. Het model is duidelijk sterk geïnspireerd op de methodologie van abstract state machines (ASMs) [31, 32] en we zullen het model dan ook aan de hand van ASMs definiëren.

We tonen aan dat de verschil- en de semijoin-operatie niet berekenbaar zijn door een FCM, zelfs niet wanneer de FCM bit strings mag opslaan met een totale lengte van $o(n)$, waarbij n de lengte van de invoerlijsten is. Wanneer echter alle gesorteerde versies van de relaties van de gegevensbank ter beschikking zijn, kan een FCM elke operator van de semijoin algebra berekenen. Bijgevolg kan elke semijoin algebra query berekend worden door een uitvoeringsplan dat bestaat uit FCMs en sorteeroperaties.

In zulke uitvoeringsplannen duiken in het algemeen echter veel tussentijdse sorteeroperaties op. In sommige gevallen kunnen die tussentijdse sorteeroperaties worden vermeden door in het begin een slimme sorteervolgorde te kiezen die dan door alle operaties in het uitvoeringsplan kan worden gebruikt [56]. Daarom rijst natuurlijk de vraag: zijn tussentijdse sorteeroperaties wel echt nodig? Of anders gezegd, kan elke semijoin algebra query al worden berekend door een FCM die werkt op gesorteerde invoerlijsten? We beantwoorden deze vraag negatief: De zeer eenvoudige semijoin algebra expressie $R \ltimes (S \ltimes T)$ met R en T unaire relaties en S een binaire relatie, is niet berekenbaar door een FCM met intern geheugen van grootte $o(n)$ die werkt op gesorteerde inputs. We tonen tevens aan dat de tussentijdse sorteeroperaties zelfs onvermijdelijk zijn bij het berekenen van semijoin algebra queries wanneer FCMs kunnen werken op combinaties van voorwaartse en achterwaartse sorteringen van de invoerlijsten. We merken daarbij op dat FCMs inderdaad meer relationele algebra queries kunnen berekenen wanneer ze kunnen werken op voorwaartse én achterwaartse sorteringen.

Met het specifieke karakter van finite cursor machines, meer bepaald de sequentiële verwerking van gegevens, in gedachten komen we in Hoofdstuk 6 bij het onderwerp van "stream query processing". Dit onderwerp heeft de laatste jaren enorm veel aandacht gekregen in de onderzoeksgemeenschap van gegevensbanksystemen. We geven hier slechts enkele referenties [57, 9, 10, 47, 20]; de publicatielijst is veel langer. Stream queries zijn typisch "continu" in de zin dat het resultaat continu moet worden aangepast als er nieuwe gegevens aankomen: stream toepassingen zijn inderdaad "data-driven". Bijgevolg moeten stream queries op een incrementele manier worden berekend. Dat gebeurt met wat men noemt "non-blocking" operatoren. De monotone relationele algebra operatoren zijn non-blocking; de operatoren die niet monotoon zijn, zoals bijvoorbeeld het verschil, of groepering en aggregatie, worden meestal non-blocking gemaakt door ze te beperken tot "sliding windows".

Eerst geven we een theoretisch raamwerk dat een aantal eerder filosofische kwesties over stream queries tracht te verhelderen. Bijvoorbeeld, als we streams zien als oneindige lijsten en we zien queries als functies van streams naar streams, wat betekent het dan dat een stream query berekenbaar is? Is berekenbaarheid hetzelfde als continuïteit? Wat is nu precies het verband tussen continuïteit en montoniciteit? Kunnen we een formele definitie geven van een non-blocking operator?

Eerder al rapporteerden Arasu en Widom [7] en Law, Wang en Zaniolo [41] over gelijkaardig werk. Dit werk heeft echter een aantal nieuwigheden:

Ten eerste maken we een onderscheid tussen "timed" en "untimed" toepassingen. In een timed toepassing zijn de tijdstippen van de uitvoer-stream gesynchroniseerd met die van de invoer-stream en in een untimed toepassing is dat niet zo. De meeste toepassingen die in de literatuur worden vermeld, zoals aandelenkoersen of sensornetwerken, zijn timed. Toch hebben untimed streams zeker en vast toepassingen, bijvoorbeeld in audio of video streams en in uitzendingen via het Internet. Daar is de logische volgorde waarin de pakketten ontvangen of verstuurd zijn veel belangrijker dan de juiste tijdstippen waarop de pakketten zijn ontvangen of verstuurd. Fundamenteler nog, we kunnen timed streams zien als een speciaal geval van untimed streams en de theorie omtrent untimed streams gebruiken om timed streams te bestuderen. Toch zullen we ook enkele eigenschappen specifiek aan timed queries

bestuderen, zoals bijvoorbeeld het niet-voorspellende karakter.

Ten tweede vinden onze formele definities van abstract berekenbare stream queries hun oorsprong in de theorie van type-2 effectivity (TTE) [60]. Dit is een theorie van berekenbaarheid op oneindige strings (en nog veel meer, maar daar gaan we hier niet op in). Het basisidee van TTE, verrassend analoog aan dat van continue stream queries, is dat willekeurig lange eindige beginwoorden van de oneindige uitvoer berekend kunnen worden door te werken op steeds langere eindige beginwoorden van de oneindige invoer. Een belangrijk inzicht van TTE is dat berekenbare functies op oneindige strings inderdaad "continu" zijn, maar dan in de betekenis van wiskundige topologie. Concreet, voor een natuurlijke metriek op oneindige strings (beter bekend als de Cantor metriek) waarbij twee strings dichter bij elkaar liggen hoe langer hun langste gemeenschappelijk beginwoord is, toont men aan dat berekenbare functies continu zijn in de standaard wiskundige betekenis. Continuïteit geeft ons op deze manier de mogelijkheid om aan te tonen dat niet zomaar elke functie van streams naar streams beschouwd kan worden als een stream query.

Ten slotte, onze theorie is abstract in de zin dat de elementen van een stream uit een willekeurig universum komen en dat daarop willekeurige predikaten en functies gedefinieerd kunnen zijn. In wiskundige logica spreekt men van een structuur en we zullen naar het universum verwijzen als de achtergrondstructuur. We gaan de stream elementen dan ook niet coderen als bit strings (eindige of oneindige) en Turing-machine operaties op deze bit strings uitvoeren. Die aspecten zijn immers al behandeld in de TTE. Bijgevolg is onze theorie erg algemeen. We tonen aan dat de berekenbare stream queries precies de continue functies zijn van streams naar streams.

We zullen argumenteren dat finite cursor machines onrealistisch krachtig zijn in het verwerken van streams. Daarom voeren we een nieuw model in en we doen dat opnieuw door gebruik te maken van de methodologie van abstract state machines [31, 32]. We noemen dat model "streaming ASM" (sASM). We tonen aan dat elke berekenbare stream query berekenbaar is door een streaming ASM met een geschikte achtergrondstructuur. Streaming ASMs laten ons bovendien toe om negatieve resultaten te bewijzen. Concreet zullen we sASMs bestuderen met een beperkt geheugen: zulke machines kunnen enkel een vast aantal elementen onthouden die eerder zijn aangekomen in de stream. In de context van query processing zijn zulke machines heel erg natuurlijk: bijvoorbeeld elke operator die werkt volgens de sliding window semantiek is berekenbaar met een beperkt geheugen. We tonen aan dat er eenvoudige queries zijn die niet berekenbaar zijn met een beperkt geheugen. Eén van de eenvoudigste queries is INTERSECT: vind de elementen die gemeenschappelijk zijn in twee streams.

Aangezien we weten dat FCMs krachtiger zijn dan sASMs en aangezien we al weten dat de query INTERSECT niet berekenbaar is door een FCM, zouden we per reductie direct kunnen besluiten dat de query ook niet berekenbaar is door een sASM. We geven hier echter een rechtstreeks bewijs. Dat heeft het voordeel — naast het feit dat het bewijs uiteraard eenvoudiger is — dat het een beter inzicht geeft in de beperkingen van stream query processing met een beperkt geheugen. We zullen bovendien ook zien dat er stream queries bestaan die berekenbaar zijn door een FCM, maar niet door een sASM.

Ten slotte, in Hoofdstuk 7 bestuderen we de uitdrukkingskracht van de semijoin

algebra wanneer er willekeurige ongekwantificeerde formules kunnen worden gebruikt in de selectie- en join-condities. Merk op dat de Codd-stelling voor de semijoin algebra in Hoofdstuk 3 enkel geldig is voor equi-semijoins.

In het eerste deel van Hoofdstuk 7 bestuderen we de kracht van het herhalen en permuteren van kolommen in de projectie-operator van de semijoin algebra. Die herhalingen en permutaties zijn ook in de relationele algebra toegestaan, maar ze zijn daar duidelijk niet essentieel. Er bestaan immers twee equivalente perspectieven over het relationeel model en over de relationele algebra, met name het benoemd perspectief en het onbenoemd perspectief [1]. In het benoemd perspectief worden tupels gezien als functies van de verzameling attributen naar het domein; in het onbenoemd perspectief worden tupels gezien als geordende lijsten van domein waarden. Het permuteren en herhalen van kolommen in de projectie-operatie is daarom enkel mogelijk in het onbenoemde perspectief. Ter volledigheid zullen we hier nog eens expliciet aantonen dat elke relationele algebra expressie kan worden herschreven als een relationele algebra expressie die geen kolommen herhaalt of permutateert in de projectie-operaties.

Voor de semijoin algebra is het echter niet direct duidelijk of het herhalen en permuteren van kolommen in de projectie-operatie invloed heeft op de uitdrukkingskracht. De herschrijfregel in het hierboven vermeld bewijs om een projectie in een RA expressie te vervangen door een projectie die noch permuteert, noch herhaalt, maakt inderdaad gebruik van de join-operatie en die operatie ontbreekt nu net in de semijoin algebra. Toch kunnen we aantonen dat elke SA expressie gesimuleerd kan worden door SA expressies waarin de projectie-operaties geen permutaties of herhalingen van kolommen gebruiken. Het begrip "simulatie" is hier echter iets ingewikkelder. Het idee is dat we vanuit een gegeven SA expressie E een verzameling SA expressies zonder herhalingen en permutaties kunnen construeren die de relevante waardes van de uitvoertupels van E als resultaat hebben. De uitvoertupels van E zelf kunnen dan worden bekomen door achteraf op de tupels van relevante waardes zekere herhalingen en permutaties uit te voeren; hoe dat dan precies moet gebeuren, kan worden afgeleid uit de vertaling. In het bijzonder, voor booleaanse expressies is er altijd een enkele equivalente booleaanse expressie zonder herhalingen en permutaties.

In een tweede deel van Hoofdstuk 7 definiëren we een Ehrenfeucht-Fraïssé spel dat de uitdrukkingskracht van de semijoin algebra karakteriseert. De karakterisatie houdt rekening met de beschikbare predikaten in de semijoin- en selectie-condities. We gebruiken het Ehrenfeucht-Fraïssé spel om in het bijzonder de uitdrukkingskracht te bestuderen van de semijoin algebra waarbij willekeurige ongekwantificeerde formules over enerzijds het vocabularium $\{=\}$ en anderzijds over het vocabularium $\{=, <\}$ kunnen worden gebruikt in selectie- en semijoin-condities.