Theoretical Computer Science 935 (2022) 105–127

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

On the expressiveness of LARA: A proposal for unifying linear and relational algebra

Pablo Barceló [a,b,c,*], Nelson Higuera [d], Jorge Pérez [e,b], Bernardo Subercaseaux [f,b]

[a] *Institute for Mathematical and Computational Engineering, PUC Chile, Chile*
[b] *IMFD Chile, Chile*
[c] *National Center for Artificial Intelligence CENIA, Chile*
[d] *Department of Informatics, Technical University of Vienna, Austria*
[e] *Department of Computer Science, University of Chile, Chile*
[f] *Computer Science Department, Carnegie Mellon University, United States of America*

ARTICLE INFO

Article history:
Received 2 September 2021
Received in revised form 21 August 2022
Accepted 5 September 2022
Available online 8 September 2022
Communicated by W. Fan

Keywords:
Languages for linear and relational algebra
Expressive power
Relational algebra with aggregation
Query genericity
Locality of queries
Safety

ABSTRACT

We study the expressive power of the LARA language – a recently proposed unified model for expressing relational and linear algebra operations – both in terms of traditional database query languages and some analytic tasks often performed in machine learning pipelines. Since LARA is parameterized by a set of user-defined functions which allow to transform values in tables, known as *extension functions*, the exact expressive power of the language depends on how these functions are defined. We start by showing LARA to be expressive complete with respect to a syntactic fragment of relational algebra with aggregation (under the mild assumption that extension functions in LARA can cope with traditional relational algebra operations such as selection and renaming). We then look further into the expressiveness of LARA based on different classes of extension functions, and distinguish two main cases depending on the level of genericity that queries are enforced to satisfy. Under strong genericity assumptions the language cannot express matrix convolution, a very important operation in current machine learning pipelines. This language is also local, and thus cannot express operations such as matrix inverse that exhibit a recursive behavior. For expressing convolution, one can relax the genericity requirement by adding an underlying linear order on the domain. This, however, destroys locality and turns the expressive power of the language much more difficult to understand. In particular, although under complexity assumptions some versions of the resulting language can still not express matrix inverse, a proof of this fact without such assumptions seems challenging to obtain.

© 2022 Elsevier B.V. All rights reserved.

1. Introduction

Background. Many of the current systems for analytics require both relational algebra and statistical functionalities for manipulating data. While tools based on relational algebra are often used for preparing and structuring the data, the ones based on statistics and machine learning (ML) are applied to quantitatively reason about such data. Based on the

* Corresponding author at: Institute for Mathematical and Computational Engineering, PUC Chile, Chile.
E-mail addresses: pbarcelo@uc.cl (P. Barceló), nelson.ruiz@tuwien.ac.at (N. Higuera), jperez@dcc.uchile.cl (J. Pérez), bsuberca@andrew.cmu.edu (B. Subercaseaux).

https://doi.org/10.1016/j.tcs.2022.09.003
0304-3975/© 2022 Elsevier B.V. All rights reserved.

"impedance mismatch" that this dichotomy creates [17], the database theory community has highlighted the need of developing a standard data model and query language for such applications, meaning an extension of relational algebra with linear algebra operators that is able to express the most common operations in ML [4,20]. Noticeably, the ML community has also recently manifested the need for what – at least from a database perspective – can be seen as a high-level language that manipulates tensors. Indeed, despite their wide adoption, there has been a recent interest in redesigning the way in which tensors are used in deep learning code [12,21,22], due to some pitfalls of the current way in which tensors are abstracted.

Hutchinson et al. [14,13] have recently proposed a data model and a query language that aims at becoming the "universal connector" that solves the aforementioned impedance. On the one hand, the proposed data model corresponds to *associative tables*, which generalize relational tables, tensors, arrays, and others. Associative tables are two-sorted, consisting of *keys* and *values* that such keys map to. The query language, on the other hand, is called Lara, and subsumes several known languages for the data models mentioned above. Lara is an algebraic language designed in a minimalistic way by only including three operators; namely, *join*, *union*, and *extension*. In rough terms, the first one corresponds to the traditional join from relational algebra, the second one to the operation of aggregation, and the third one to the extension defined by a function as in a *flatmap* operation. It has been shown that Lara subsumes all relational algebra operations and is capable of expressing several interesting linear algebra operations used in graph algorithms [13].

Our results. Based on the proposal of Lara as a unified language for relational and linear algebra, it is relevant to develop a deeper understanding of its expressive power, both in terms of the logical query languages traditionally studied in database theory, and the ML operations often performed in practical applications. We start with the former and show that Lara is expressive complete with respect to a syntactic fragment of *relational algebra with aggregation* (RA_{Agg}), a language that has been studied as a way to abstract the expressive power of SQL without recursion; cf., [18,19]. To be more precise, under the mild assumption that extension functions in Lara can cope with traditional relational algebra operations such as selection and renaming, then Lara is expressive complete with respect to a suitable syntactic fragment of RA_{Agg} under multiset semantics that ensures that expressions get properly evaluated over associative tables. The use of the multiset semantics is relevant for capturing some features from Lara that were originally defined to handle bags. While the basic idea behind this proof is simple, the details are cumbersome. The process involves, in addition, several delicate design decisions about the features included in the languages. The goal is to keep a reasonable balance between the level of generality for the result presented, on the one hand, and the simplicity and readability of the presentation, on the other hand.

Our expressive completeness result is parameterized by the class of functions, denoted by Ω, allowed in the extension operator. For each such an Ω we allow RA_{Agg} to contain all built-in predicates encoding functions in Ω. To understand which ML operators Lara can express, one then needs to bound the class Ω of extension functions allowed in the language. We start with a tame class that can still express several relevant functions. These are the RA-expressible functions that allow to compute arbitrary numerical predicates on values, but can only compare keys with respect to (in)equality. This restriction makes the logic quite amenable for theoretical exploration. In fact, it is easy to show that the resulting "tame version" of Lara satisfies a strong *genericity* criterion (in terms of key-permutations) and is also *local*, in the sense that queries in the language can only see up to a fixed-radius neighborhood from its free variables; cf., [19]. The first property implies that this tame version of Lara cannot express non-generic operations, such as matrix *convolution*, and the second one that it cannot express inherently recursive queries, such as matrix *inverse*. Both operations are very relevant for ML applications; e.g., matrix convolution is routinely applied in dimension-reduction tasks, while matrix inverse is used for example to compute the coefficients of a linear regression through the least-squares estimation.

We then look more carefully at the case of matrix convolution, and show that this query can be expressed if we relax the genericity properties of the language by assuming the presence of a linear order on the domain of keys. This relaxation implies that queries expressible in the resulting version of Lara are no longer invariant with respect to key-permutations. This language, however, is much harder to understand in terms of its expressive power. In particular, it can express non-local queries, and hence we cannot apply standard locality techniques to show that the matrix inversion query is not expressible in it. To prove this result, then, one could apply techniques based on the *Ehrenfeucht-Fraïssé* games [19] that characterize the expressive power of the logic. However, it is combinatorially difficult to show results based on such games when in the presence of a linear order. In turn, it is possible to obtain that matrix inversion is not expressible in a natural restriction of our language under computational complexity assumptions. This is because the data complexity of queries expressible in such a restricted language is Logspace, but matrix inversion is complete for a class that is believed to be a proper extension of the latter.

Discussion. The LARA language proposal emanates from the DB community as a way to capture, in a minimal way, the operations used in systems that combine analytical and numerical tasks. The main objective of our paper is understanding, then, how the expressive power of this language relates traditional database theory concepts and the arsenal of techniques that have been developed in this area to study the expressiveness of query languages. We believe that our results shed light on some important questions regarding the expressive power of Lara as explained next:

- The aforementioned expressive completeness result can be seen as a sanity check for Lara. In fact, this language is specifically tailored to handle aggregation in conjunction with relational algebra operations over associative tables, and the results in the paper show that Lara and RA_{Agg} are in fact equivalent when expressions in the latter language are

forced to be evaluated over associative tables. We observe that while LARA consists of *positive* algebraic operators only, difference over associative tables can be encoded in the language by a combination of aggregate operators and extension functions (which resembles, as expected, well-known tricks for expressing relational algebra set difference in terms of aggregation [8]).

- The expressive power as RA_{Agg} is tightly related to the expressiveness of SQL [18]. One might wonder then why to use LARA instead of SQL, since after all LARA seems to be just a version of SQL geared toward associative tables. It is difficult to give a definite answer to this question, since there seem to be arguments for both sides. On the one hand, LARA is especially tailored to deal with ML objects, such as matrices or tensors, which are naturally modeled as associative tables. As the proof of Theorem 2 suggests, in turn, RA_{Agg} requires to be trimmed in order to maintain the "key-functionality" of associative tables. On the other hand, LARA was not designed as a declarative language, and this is evidenced by observing, for example, how difficult it is to express simple ML tasks like matrix convolution in LARA (see Section 7).
- The expressive power of LARA is strongly influenced by the class of extension functions one is allowed to use in the language. A weak class of extension functions keeps us in the realm of RA_{Agg} (without extension functions!), which is probably not what is desired. In turn, a rich class of extension functions might allow to express powerful properties but possibly at the cost of increasing the computational complexity of evaluation and turning more difficult our capacity to understand the expressiveness of the language.
- Recently, it has been strongly argued for the need to embed the ability to express ML operations in the foundation of relational algebra and SQL [16]. Our results can also be seen as a contribution in this regard, as they help elucidate which of these functionalities can be expressed in these languages, either directly or based on specific built-in relations that could be embedded in them.

Related work. MATLANG [2,6] is a pure matrix-manipulation language based on elementary linear algebra operations. It is shown that this language is contained in the three-variable fragment of relational algebra with summation and, thus, it is local. This implies that the core of MATLANG cannot check for the presence of a four-clique in a graph (represented as a Boolean matrix), as this query requires at least four variables to be expressed, and neither it can express the non-local matrix inversion query. It can be shown that MATLANG is strictly contained in the tame version of LARA that is mentioned above. A recent and particularly interesting line of work proposes an extension of MATLANG with recursive features [7]. This extension subsumes the class of arithmetic circuits, which is often said to capture linear algebra. Linking the expressive power of this language with the proposal we make here for extending LARA with a linear order on keys seems to be a relevant, but challenging, direction for future work.

Related version of the paper. This is the full version of a paper, with a similar title, originally published at the International Conference on Database Theory, ICDT, in 2020 [1]. In addition to providing full proofs to the theoretical results, which were not present in the ICDT version, the current version also includes more examples and graphical explanations of the concepts introduced. More importantly, the current version clarifies and presents in a cleaner way all the assumptions needed for the main expressive completeness result to hold. This result is now presented in terms of named relational algebra, instead of first-order logic as in the original ICDT paper, as suggested by the reviewers of this submission. We are very grateful to the reviewers for this suggestion, which clearly helped us improved the readability of the result.

Organization of the paper. Basics of LARA and RA_{Agg} are presented in Sections 2 and 3, respectively. All the assumptions needed for our expressive completeness result to hold are stated in Section 4. The expressive completeness of LARA in terms of RA_{Agg} is shown in Section 5. The tame version of LARA and some inexpressibility results relating to it are given in Section 6, while in Section 7 we present a version of LARA that can express convolution and some discussion about its expressive power. We finalize in Section 8 with concluding remarks and future work.

2. The LARA language

In this section we introduce the data model, syntax, and semantics used by the LARA language as defined by Hutchinson et al. [14].

2.1. Basics

For integers $m \leq n$, we write $[m, n]$ for $\{m, \dots, n\}$ and $[n]$ for $\{1, \dots, n\}$. If $\bar{v} = (v_1, \dots, v_n)$ is a tuple of elements, we write $\bar{v}[i]$ for v_i. We denote multisets as $\{\!\{a, b, \dots\}\!\}$.

An *aggregate operator* over a set U is a partial function $\oplus$ that takes a multiset of k elements from U, for $k \leq \omega$, and returns a single element from U. This notion generalizes most aggregate operators used in practical query languages; e.g., SUM, MIN, MAX, AVG, and COUNT. We assume that each operator $\oplus$ over U has a *neutral value* $0_\oplus \in U$. Formally, $\oplus(W) = \oplus(W')$, for every multiset W of elements in U and every extension W' of W with an arbitrary number of occurrences of $0_\oplus$. As an example, for SUM the neutral value is 0, and for MIN and MAX the neutral values are $-\infty$ and ∞, respectively.

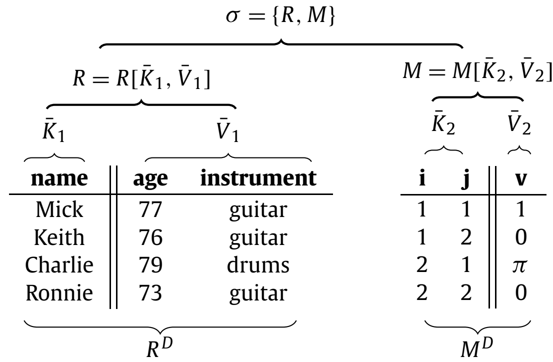

Fig. 1. Illustration of the basic elements of the Lara language. The *relational schema* σ contains two *relation symbols*: R and M. While R represents a table of musicians, M represents a matrix with real coefficients. Both Keys and Values correspond to $\Sigma^* \cup \mathbb{R}$, where Σ is the English alphabet, as keys and values can take alphanumeric values.

For operators $\oplus$ such as AVG and COUNT, that do not have a natural neutral value, we can simply assume that $0_\oplus$ is a special distinguished symbol in $U \setminus \mathbb{R}$ for which the following holds for each multiset W over $\mathbb{R} \cup \{0_\oplus\}$:

$$\oplus(W) \;=\; \oplus(W \cap \mathbb{R}).$$

For simplicity, we see commutative binary operations $\otimes$ on U as aggregate operators $\oplus$ which are only defined on multisets with two elements. In this case we write $u \otimes v$ instead of the more cumbersome $\otimes(\{\{u, v\}\})$.

2.2. Data model used by LARA

A *relational schema* is a finite collection σ of *two-sorted* relation symbols. The first sort consists of *key-attributes* and the second one of *value-attributes*. Each relation symbol $R \in \sigma$ is then associated with a pair $(\bar{K}, \bar{V})$, where $\bar{K}$ and $\bar{V}$ are nonempty sets of key- and value-attributes, respectively. We often write $R[\bar{K}, \bar{V}]$ to denote that $(\bar{K}, \bar{V})$ is the *sort* of R.

There are two countably infinite sets of objects that populate databases: a domain of *keys*, which interpret key-attributes and is denoted Keys, and a domain of *values*, which interpret value-attributes and is denoted Values. A Lara-*tuple of sort* $(\bar{K}, \bar{V})$ is a function $t : \bar{K} \cup \bar{V} \to \mathsf{Keys} \cup \mathsf{Values}$ such that $t(A) \in \mathsf{Keys}$ if $A \in \bar{K}$ and $t(A) \in \mathsf{Values}$ if $A \in \bar{V}$. We naturally identify a Lara-tuple with a regular tuple by assuming an ordering of the attributes. For example, if $\bar{K} = \{\mathbf{i}, \mathbf{j}\}$ and $\bar{V} = \{\mathbf{v}\}$, a possible Lara-tuple is the mapping $\mathbf{i} \to 1, \mathbf{j} \to 0, \mathbf{v} \to 7$, which can be naturally identified with the tuple $(1, 0, 7)$ by assuming the order $\mathbf{i}, \mathbf{j}, \mathbf{v}$. As the order of attributes is not relevant in Lara, and yet it is often convenient to refer to Lara-tuples as regular tuples, we will simply assume an implicit ordering of the attributes.

A *database* D over schema σ is a mapping that assigns with each relation symbol $R[\bar{K}, \bar{V}] \in \sigma$ a finite set R^D of tuples of sort $(\bar{K}, \bar{V})$. We often see D as a set of *facts*, i.e., expressions $R(t)$ with $t \in R^D$. For ease of presentation, we write $R(\bar{k}, \bar{v}) \in D$ if $R(t) \in D$ for some t with $t(\bar{K}) = \bar{k}$ and $t(\bar{V}) = \bar{v}$ with $\bar{k} \in \mathsf{Keys}^{|\bar{K}|}$ and $\bar{v} \in \mathsf{Values}^{|\bar{V}|}$. Fig. 1 illustrates these concepts.

For a database D to be a Lara *database* we need D to satisfy an extra restriction: Key attributes in fact define a key constraint over the corresponding relation symbols. That is,

$$R(\bar{k}, \bar{v}), R(\bar{k}, \bar{v}') \in D \quad \Longrightarrow \quad \bar{v} = \bar{v}',$$

for each $R[\bar{K}, \bar{V}] \in \sigma$, $\bar{k} \in \mathsf{Keys}^{|\bar{K}|}$, and $\bar{v}, \bar{v}' \in \mathsf{Values}^{|\bar{V}|}$. Relations of the form R^D that satisfy this constraint are called *associative tables* [14]. Yet, we abuse terminology and call associative table to any set A of tuples of the same sort $(\bar{K}, \bar{V})$ such that $\bar{v} = \bar{v}'$ for each $(\bar{k}, \bar{v}), (\bar{k}, \bar{v}') \in A$. In such a case, A is of sort $(\bar{K}, \bar{V})$. Notice that for a tuple $(\bar{k}, \bar{v})$ in A, we can safely denote $\bar{v} = A(\bar{k})$.

2.3. Syntax of LARA

The Lara language makes extensive use of the class of *extension functions*, as defined next.

Definition 1 *(Extension function).* An *extension function* f *of sort* $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$ maps each tuple t of sort $(\bar{K}, \bar{V})$ to a finite associative table of sort $(\bar{K}', \bar{V}')$, for $\bar{K} \cap \bar{K}' = \emptyset$ and $\bar{V} \cap \bar{V}' = \emptyset$.

As an example, an extension function might take a tuple $t = (k, v_1, v_2)$ of sort (K, V_1, V_2), for $v_1, v_2 \in \mathbb{Q}$, and map it to a table of sort (K', V') that contains a single tuple (k, v), where v is the average between v_1 and v_2.

The syntax of Lara is parameterized by a set Ω of user-defined extension functions. The syntax of the resulting language is given next.

i	**j**	**v**$_1$	**v**$_2$
0	0	1	5
0	1	2	6
1	0	3	7
1	1	4	8

j	**k**	**v**$_2$	**v**$_3$
0	0	1	1
0	1	1	2
1	0	1	1
1	1	2	1

Fig. 2. Associative tables A and B used as examples when defining the semantics of Lara.

Definition 2 *(LARA language).* We inductively define the set of expressions in Lara(Ω) over schema σ as follows.

- **Empty associative table.** There is an expression $\emptyset$ of sort $(\emptyset, \emptyset)$.
- **Atomic expressions.** If $R[\bar{K}, \bar{V}]$ is in σ, then R is an expression of sort $(\bar{K}, \bar{V})$.
- **Join.** If e_1 and e_2 are expressions of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$, respectively, and $\otimes$ is a commutative binary operator over Values, then $e_1 \bowtie_{\otimes} e_2$ is an expression of sort $(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$.
- **Union.** If e_1, e_2 are expressions of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$, respectively, and $\oplus$ is an aggregate operator over Values, then $e_1 \between_{\oplus} e_2$ is an expression of sort $(\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$.
- **Extend.** For e an expression of sort $(\bar{K}, \bar{V})$ and f a function in Ω of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, it is the case that $\mathsf{Ext}_f\, e$ is an expression of sort $(\bar{K} \cup \bar{K}', \bar{V}')$.

We often write $e[\bar{K}, \bar{V}]$ to denote that the expression e is of sort $(\bar{K}, \bar{V})$.

2.4. Semantics of LARA

We introduce some important notions before we give the definition of the semantics of Lara.

Padding. Let $\bar{V}_1$ and $\bar{V}_2$ be tuples of value-attributes, and $\bar{v}$ a tuple over Values of sort $\bar{V}_1$. Then $\mathrm{pad}_{\oplus}^{\bar{V}_2}(\bar{v})$ is a new tuple $\bar{v}'$ over Values of sort $\bar{V}_1 \cup \bar{V}_2$ such that for each $V \in \bar{V}_1 \cup \bar{V}_2$ we have that $v'(V) = v(V)$, if $V \in \bar{V}_1$, and $v'(V)$ is the neutral value $0_{\oplus}$ for $\oplus$, otherwise.

Compatible tuples. Consider tuples $\bar{k}_1$ and $\bar{k}_2$ over key-attributes $\bar{K}_1$ and $\bar{K}_2$, respectively. We say that $\bar{k}_1$ and $\bar{k}_2$ are *compatible*, if $\bar{k}_1(K) = \bar{k}_2(K)$ for every $K \in \bar{K}_1 \cap \bar{K}_2$. If $\bar{k}_1$ and $\bar{k}_2$ are compatible, one can then naturally define the extended tuple $\bar{k}_1 \cup \bar{k}_2$ over key-attributes $\bar{K}_1 \cup \bar{K}_2$. Also, given a tuple $\bar{k}$ of sort $\bar{K}$, and a set $\bar{K}' \subseteq \bar{K}$, the restriction $\bar{k}{\downarrow}_{\bar{K}'}$ of $\bar{k}$ to attributes $\bar{K}'$ is the only tuple of sort $\bar{K}'$ that is compatible with $\bar{k}$. These notions are defined analogously for tuples over value-attributes.

Solve operator. Finally, given a multiset T of Lara-tuples $(\bar{k}, \bar{u})$ of the same sort $(\bar{K}, \bar{V})$, where $|\bar{V}| = m$, we define $\mathsf{Solve}_{\oplus}(T)$ as

$$\{(\bar{k}, \bar{v}) \mid \text{there exists } \bar{u} \text{ such that } (\bar{k}, \bar{u}) \in T \text{ and } \bar{v}[i] = \bigoplus_{\bar{v}' : (\bar{k}, \bar{v}') \in T} \bar{v}'[i], \text{for each } 1 \leq i \leq m\}.$$

That is, $\mathsf{Solve}_{\oplus}(T)$ turns the multiset T into an associative table T' by first grouping together tuples that have the same value over $\bar{K}$, and the solving key-conflicts by aggregating with respect to $\oplus$ (coordinate-wise).

The evaluation of a Lara(Ω) expression e over a schema σ on a Lara database D, denoted e^D, is inductively defined next. For the reader's convenience, we also summarize these definitions in Fig. 3. Since the definitions are not so easy to grasp, we use the associative tables A and B in Fig. 2 to construct examples. Here, **i**, **j**, and **k** are key-attributes, while **v**$_1$, **v**$_2$, and **v**$_3$ are value-attributes. In the definition below we use $A(\bar{k})$, for an associative table $A[\bar{K}, \bar{V}]$ and a tuple $\bar{k}$ in the projection of A over $\bar{K}$, to denote the unique tuple $\bar{v} \in \mathsf{Values}^{|\bar{V}|}$ such that $(\bar{k}, \bar{v}) \in A$.

Empty associative table. If $e = \emptyset$, then e^D is simply an empty associative table of sort $(\emptyset, \emptyset)$.

Atomic expressions. If $e = R[\bar{K}, \bar{V}]$, for $R \in \sigma$, then we have that $e^D := R^D$.

Join. If $e[\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = e_1[\bar{K}_1, \bar{V}_1] \bowtie_{\otimes} e_2[\bar{K}_2, \bar{V}_2]$, then

$$e^D := \{(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \otimes \bar{v}_2) \mid \bar{k}_1 \text{ and } \bar{k}_2 \text{ are compatible, } \bar{v}_1 = \mathrm{pad}_{\otimes}^{\bar{V}_2}(e_1^D(\bar{k}_1)),\ \bar{v}_2 = \mathrm{pad}_{\otimes}^{\bar{V}_1}(e_2^D(\bar{k}_2))\}.$$

Here, $\bar{v}_1 \otimes \bar{v}_2$ abbreviates

$$(\bar{v}_1[1] \otimes \bar{v}_2[1], \ \ldots \ \bar{v}_1[n] \otimes \bar{v}_2[n]),$$

assuming that $|\bar{v}_1| = |\bar{v}_2| = n$.

For example, the result of $A \bowtie_{\times} B$ is shown in Fig. 4, for $\times$ being the usual product on $\mathbb{N}$ and $0_{\times} = 1$.

	Expression e	**Evaluation** e^D
Empty associative table	$\emptyset$	$\emptyset$
Atomic expression	$R[\bar{K}, \bar{V}]$	R^D
Join	$e_1 \bowtie_{\otimes} e_2$	$\{(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \otimes \bar{v}_2) \mid$ $\bar{k}_1$ and $\bar{k}_2$ are compatible, $\bar{v}_1 = \text{pad}_{\otimes}^{\bar{V}_2}(e_1^D(\bar{k}_1))$, and $\bar{v}_2 = \text{pad}_{\otimes}^{\bar{V}_1}(e_2^D(\bar{k}_2))\}$
Union	$e_1 \rotatebox{90}{$\bowtie$}_{\oplus} e_2$	$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in e_1^D,\ \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}$ and $\bar{v} = \text{pad}_{\oplus}^{\bar{V}_2}(\bar{v}_1)$, or $(\bar{k}_2, \bar{v}_2) \in e_2^D,\ \bar{k} = \bar{k}_2{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}$ and $\bar{v} = \text{pad}_{\oplus}^{\bar{V}_1}(\bar{v}_2)\}\!\}$
Extend	$\mathsf{Ext}_f\, e_1$	$\{(\bar{k} \cup \bar{k}', \bar{v}') \mid (\bar{k}, \bar{v}) \in e_1^D$ and $(\bar{k}', \bar{v}') \in f(\bar{k}, \bar{v})\}$
Map	$\mathsf{Map}_f\, e_1$	$\{(\bar{k}, \bar{v}') \mid (\bar{k}, \bar{v}) \in e_1^D$ and $\bar{v}' \in f(\bar{k}, \bar{v})\}$
Aggregate	$\rotatebox{90}{$\bowtie$}_{\oplus}^{\bar{K}}\, e_1$	$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}}$ and $\bar{v} = e_1^D(\bar{k}_1)\}\!\}$
Reduce	$\bar{\rotatebox{90}{$\bowtie$}}_{\oplus}^{\bar{K}}\, e_1$	$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid \bar{k} = \bar{k}_1{\downarrow}_{\bar{L} \setminus \bar{K}}$ and $\bar{v} = e_1^D(\bar{k}_1)\}\!\}$, for e_1 of sort $(\bar{L}, \bar{V})$

Fig. 3. Summary of the syntax and semantics of Lara.

Union. If $e[\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = e_1[\bar{K}_1, \bar{V}_1] \rotatebox{90}{$\bowtie$}_{\oplus} e_2[\bar{K}_2, \bar{V}_2]$, then e^D is

$$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid \text{(a)}\,(\bar{k}_1, \bar{v}_1) \in e_1^D,\ \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}_1 \cap \bar{K}_2},\ \text{and } \bar{v} = \text{pad}_{\oplus}^{\bar{V}_2}(\bar{v}_1),\ \text{or}$$
$$\text{(b)}\,(\bar{k}_2, \bar{v}_2) \in e_2^D,\ \bar{k} = \bar{k}_2{\downarrow}_{\bar{K}_1 \cap \bar{K}_2},\ \text{and } \bar{v} = \text{pad}_{\oplus}^{\bar{V}_1}(\bar{v}_2)\}\!\}.$$

In more intuitive terms, e^D is defined by first projecting over $\bar{K}_1 \cap \bar{K}_2$ any tuple in e_1^D, resp., in e_2^D. As the resulting multiset of tuples might no longer be an associative table (because there might be many tuples with the same keys), we have to solve the conflicts by applying the given aggregate operator $\oplus$. This is what $\mathsf{Solve}_{\oplus}$ does.

For example, the result of $A \rotatebox{90}{$\bowtie$}_{+} B$ is shown in Fig. 4, for $+$ being the addition on $\mathbb{N}$.

Extend. If $e[\bar{K} \cup \bar{K}', \bar{V}'] = \mathsf{Ext}_f\, e_1[\bar{K}, \bar{V}]$ and f is of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, then

$$e^D := \{(\bar{k} \cup \bar{k}', \bar{v}') \mid (\bar{k}, \bar{v}) \in e_1^D \text{ and } (\bar{k}', \bar{v}') \in f(\bar{k}, \bar{v})\}.$$

Notice that in this case $\bar{k} \cup \bar{k}'$ always exists, unless $f(\bar{k}, \bar{v})$ is the empty associative table, as $\bar{K} \cap \bar{K}' = \emptyset$. When $f(\bar{k}, \bar{v})$ is the empty associative table, then e^D contains no tuple of the form $(\bar{k}, \cdots)$.

As an example, Fig. 4 shows the result of $\mathsf{Ext}_g\, A$, where g is a function that does the following: If the key corresponding to attribute $\mathbf{i}$ is 0, then the tuple is associated with the associative table of sort $(\emptyset, \mathbf{z})$ containing only the tuple $(\emptyset, 1)$. Otherwise, the tuple is associated with the empty associative table.

Several useful operators, as described below, can be derived from the previous ones.

- **Map operation.** An important particular case of Ext_f occurs when f is of sort $(\bar{K}, \bar{V}) \mapsto (\emptyset, \bar{V}')$, i.e., when f does not extend the keys in the associative table but only modifies the values. Following [14], we write this operation as Map_f.
- **Aggregation.** This corresponds to an aggregation over some of the key-attributes of an associative table. Consider a Lara expression $e_1[\bar{K}_1, \bar{V}_1]$, an aggregate operator $\oplus$ over Values, and a $\bar{K} \subseteq \bar{K}_1$, then $e = \rotatebox{90}{$\bowtie$}_{\oplus}^{\bar{K}}\, e_1$ is an expression of sort $(\bar{K}, \bar{V}_1)$ such that

  $$e^D := \mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}} \text{ and } \bar{v} = e_1^D(\bar{k}_1)\}\!\}.$$

  We note that $\rotatebox{90}{$\bowtie$}_{\oplus}^{\bar{K}}\, e_1$ is equivalent to the expression $e_1 \rotatebox{90}{$\bowtie$}_{\oplus} \mathsf{Ext}_f(\emptyset)$, where f is the function that associates an empty table of sort $(\bar{K}, \emptyset)$ to every possible tuple.
- **Reduction.** The reduction operator, denoted by $\bar{\rotatebox{90}{$\bowtie$}}$, is just a syntactic variation of aggregation defined as $\bar{\rotatebox{90}{$\bowtie$}}_{\oplus}^{\bar{L}}\, e_1 \equiv \rotatebox{90}{$\bowtie$}_{\oplus}^{\bar{K} \setminus \bar{L}}\, e_1$, assuming that e_1 is of sort $(\bar{K}, \bar{V})$.

Table $A \bowtie_{\times} B$

i	**j**	**k**	$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$
0	0	0	1	5	1
0	0	1	1	5	2
0	1	0	2	6	1
0	1	1	2	12	1
1	0	0	3	7	1
1	0	1	3	7	2
1	1	0	4	8	1
1	1	1	4	16	1

Table $A \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}_{+} B$

j	$\mathbf{v}_1$	$\mathbf{v}_2$	$\mathbf{v}_3$
0	4	14	3
1	8	17	2

Table $\mathrm{Ext}_g\ A$

i	**j**	**z**
0	0	1
0	1	1

Fig. 4. The tables $A \bowtie_{\times} B$, $A \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}_{+} B$, and $\mathrm{Ext}_f\ A$.

Next we provide an example that applies several of these operators.

Example 1. Consider the schema

$$\mathsf{Temp}[(\textit{time}, \textit{sensor}, \textit{object}), (\textit{temp})],$$

which represents the measurement of the temperature of different objects, as measured by different sensors, over time. The key attributes for this schema are *time*, *sensor*, and *object*, while *temp* is the unique value attribute. This schema is interpreted as an associative table A, which contains tuples of the form $\big((t, s, o), m\big)$ indicating that m is the temperature measured by sensor s for object o at time t.

Assume that, in order to take a decision, one wants to first obtain for every sensor the maximum temperature of every object that has been measured by it over the time steps, and then apply a normalization function such as *softmax*.[1] One can specify the entire process in Lara as follows.

$$\begin{aligned} \mathrm{Max} &= \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}^{\overline{(\textit{time})}}_{\max(\cdot)}\ \mathsf{Temp} && (1)\\ \mathrm{Exp} &= \mathsf{Map}_{\exp(\cdot)}\mathrm{Max} && (2)\\ \mathrm{SumExp} &= \mathbin{\rotatebox[origin=c]{90}{$\bowtie$}}^{(\textit{sensor})}_{\mathrm{sum}(\cdot)}\ \mathrm{Exp} && (3)\\ \mathrm{Softmax} &= \mathrm{Exp} \bowtie_{\div} \mathrm{SumExp} && (4) \end{aligned}$$

Expression (1) performs a reduction over the *time* attribute to obtain the new associative table A_1 over schema $\mathsf{Max}[(\textit{sensor}, \textit{object}), (\textit{temp})]$. Associative table A_1 contains all tuples of the form $\big((s, o), m\big)$ such that

$$m = \max\{m' \mid \big((t, s, o), m'\big) \in A\}.$$

Expression (2) applies a point-wise exponential function over A_1 to obtain an associative table A_2 over schema $\mathsf{Exp}[(\textit{sensor}, \textit{object}), (\textit{exptemp})]$, where *exptemp* is a fresh attribute value, such that A_2 contains all tuples $\big((s, o), e^m\big)$ with $\big((s, o), m\big) \in A_1$.

In expression (3) we apply an aggregation to compute the sum of the exponentials of all the maximum temperatures reached by objects over a single sensor. In this way we obtain an associative table A_3 over a new schema $\mathsf{SumExp}[(\textit{sensor}), (\textit{exptemp})]$, such that A_3 contains all tuples of the form (s, a) with $a = \sum_{((s,o),e^m)\in A_2} e^m$.

Finally, the expression given in (4) applies point-wise division over the associative tables A_2 and A_3. This defines a final associative table A_4 over schema $\mathsf{Softmax}[(\textit{sensor}, \textit{object}), (\textit{normtemp})]$, where *normtemp* is a fresh attribute value, which contains all tuples of the form $\big((s, o), e^m/a\big)$ such that $\big((s, o), e^m\big) \in A_2$ and $(s, a) \in A_3$. In this way we obtain our desired result. □

[1] Recall that for a vector $\bar{v} = (v_1, \ldots, v_n)$ over $\mathbb{R}$, the result of applying a *softmax* normalization over it is the vector $\bar{w} = (w_1, \ldots, w_n)$, where $w_i = \frac{e^{v_i}}{\sum_{j\le n} e^{v_j}}$.

2.5. Safety of LARA

It is easy to see that for each LARA expression e and LARA database D, the result $e(D)$ is always an associative table. Moreover, although the elements in the evaluation $e(D)$ of an expression e over D are not necessarily in D (due to the applications of the operator $\mathsf{Solve}_{\oplus}$ and the extension functions in Ω), all LARA expressions are *safe*, i.e., the cardinality of e^D is finite.

Proposition 1. *Let e be a* LARA(Ω) *expression. Then e^D is a finite associative table, for every* LARA *database D.*

3. Relational algebra with aggregation

In this paper, we study the relative expressive power of LARA in terms of a simple version of *named* RA with aggregation, following the presentation given by Libkin [18]. As for the case of LARA, the expressions in RA with aggregation consist of key- and value-attributes which are interpreted over Keys and Values, respectively. As before, we denote key-attributes as $K, K', K_1, \ldots$ and value-attributes as $V, V', V_1, \ldots$.

Syntax of RA with aggregation. We consider a vocabulary σ of two-sorted relations. In order to cope with the demands of the extension functions used by LARA (as explained later), we allow the language to be parameterized by a collection Ψ of user-defined *built-in* relations R of some sort $(\bar{K}, \bar{V})$. For each $R \in \Psi$ we blur the distinction between the symbol R and its interpretation over $\mathsf{Keys}^{|\bar{K}|} \times \mathsf{Values}^{|\bar{V}|}$. These built-in relations replace the *function-application* features that can often be encountered in the versions of RA with aggregation studied in the literature.

Definition 3 *(RA with aggregation).* Let Ψ be a set of relations R as defined above. The set of expressions in the language $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ over schema σ is inductively defined as follows:

- ***(Atomic expressions).*** Atomic expressions of $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ are $\perp$, which is of sort $\emptyset$, and R, for $R \in \sigma \cup \Psi$ of sort $(\bar{K}, \bar{V})$.
- ***(Renaming).*** If ϕ is an expression of sort $(\bar{K}_1, \bar{V}_1)$, $\bar{K} \subseteq \bar{K}_1$, and $\bar{V} \subseteq \bar{V}_1$, then $\rho_{\bar{K}\to\bar{K}', \bar{V}\to\bar{V}'}(\phi)$ is an expression of sort $(\bar{K}_1 \setminus \bar{K}, \bar{K}', \bar{V}_1 \setminus \bar{V}, \bar{V}')$, assuming that $\bar{K}_1 \cap \bar{K}' = \bar{V}_1 \cap \bar{V}' = \emptyset$, $|\bar{K}| = |\bar{K}'|$, and $|\bar{V}| = |\bar{V}'|$.
- ***(Join).*** If ϕ, ϕ' are expressions of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$, respectively, then $\phi \bowtie \phi'$ is an expression of sort $(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$.
- ***(Selection).*** If ϕ is an expression of sort $(\bar{K}, \bar{V})$, then $\sigma_{K_1 \theta K_2}\phi$ and $\sigma_{V_1 \theta V_2}\phi$ are expressions of sort $(\bar{K}, \bar{V})$, assuming that $K_1, K_2 \in \bar{K}$, $V_1, V_2 \in \bar{V}$, and $\theta \in \{=, \neq\}$.
- ***(Difference and sum).*** If ϕ, ϕ' are expressions of sort $(\bar{K}, \bar{V})$, then $\phi \setminus \phi'$ and $\phi + \phi'$ are expressions of sort $(\bar{K}, \bar{V})$.
- ***(Projection).*** If ϕ is an expression of sort $(\bar{K}, \bar{V})$, then $\pi_{\bar{K}', \bar{V}'}\phi$ is an expression of sort $(\bar{K}', \bar{V}')$, assuming $\bar{K}' \subseteq \bar{K}$ and $\bar{V}' \subseteq \bar{V}$.
- ***(Grouping).*** If ϕ is an expression of sort $(\bar{K}, \bar{V})$ and $\oplus$ is an aggregate operator over Values, then $\mathsf{Group}_{\oplus}\phi$ is an expression of sort $(\bar{K}, \bar{V})$.

We write $\phi(\bar{K}, \bar{V})$ to denote that ϕ is of sort $(\bar{K}, \bar{V})$.

Semantics of RA with aggregation. As mentioned before, for the results in the paper to hold we need the semantics of $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ to be based on multisets. In the following, if T is a multiset of tuples of sort $(\bar{K}, \bar{V})$ and t a tuple of sort $(\bar{K}, \bar{V})$, we write $\#(T, t)$ for the number of times the tuple t appears in T.

To evaluate a $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ expression, we are given a database D which interprets each symbol $R(\bar{K}, \bar{V}) \in \sigma \cup \Psi$ as a multiset R^D of facts of sort $(\bar{K}, \bar{V})$. The evaluation of an $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ expression ϕ over D, denoted ϕ^D, is then a multiset of facts which is inductively defined as follows:

- If $\phi = \perp$, then $\phi^D := \emptyset$. If $\phi = R(\bar{K}, \bar{V})$, then $\phi^D := R^D$.
- If $\phi = \rho_{\bar{K}\to\bar{K}', \bar{V}\to\bar{V}'}(\phi_1)$, then ϕ^D is the same as ϕ_1^D but with the sort changed by renaming key attributes in $\bar{K}$ to $\bar{K}'$ and value-attributes in $\bar{V}$ to $\bar{V}'$.
- If $\phi(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2) = \phi_1(\bar{K}_1, \bar{V}_1) \bowtie \phi_2(\bar{K}_2, \bar{V}_2)$, then ϕ^D is the multiset that contains precisely the tuples t of the form $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \cup \bar{v}_2)$ for which $(\bar{k}_1, \bar{v}_1) \in \phi_1^D$, $(\bar{k}_2, \bar{v}_2) \in \phi_2^D$, $\bar{k}_1$ and $\bar{k}_2$ are compatible over $\bar{K}_1 \cup \bar{K}_2$, and $\bar{v}_1$ and $\bar{v}_2$ are compatible over $\bar{V}_1 \cup \bar{V}_2$. Moreover, for a tuple $t \in \phi^D$ of the form $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \cup \bar{v}_2)$ we have that
$$\#(\phi^D, t) \;=\; \#(\phi_1^D, (\bar{k}_1, \bar{v}_1)) \cdot \#(\phi_2, (\bar{k}_2, \bar{v}_2)).$$
- If $\phi(\bar{K}, \bar{V}) = \sigma_{K_1 \theta K_2}\phi_1$, then ϕ^D is the multiset that satisfies the following for each tuple t of sort $(\bar{K}, \bar{V})$: $\#(\phi^D, t) = \#(\phi_1^D, t)$, if $t(K_1)\,\theta\,t(K_2)$, and $\#(\phi^D, t) = 0$ otherwise. The evaluation of $\sigma_{V_1 \theta V_2}\phi_1$ is defined analogously.
- If $\phi(\bar{K}, \bar{V}) = \phi_1(\bar{K}, \bar{V}) \setminus \phi_2(\bar{K}, \bar{V})$, then ϕ^D is the multiset of facts that satisfies
$$\#(\phi^D, t) \;=\; \max\,(\#(\phi_1^D, t) - \#(\phi_2^D, t), 0),$$

for each tuple t of sort $(\bar{K}, \bar{V})$.

- If $\phi(\bar{K}, \bar{V}) = \phi_1(\bar{K}, \bar{V}) + \phi_2(\bar{K}, \bar{V})$, then ϕ^D is the multiset of facts that satisfies

$$\#(\phi^D, t) \ = \ \#(\phi_1^D, t) + \#(\phi_2^D, t),$$

for each tuple t of sort $(\bar{K}, \bar{V})$.

- If $\phi(\bar{K}, \bar{V}) = \pi_{\bar{K},\bar{V}}\phi_1(\bar{K}_1, \bar{V}_1)$, then ϕ^D is the multiset of facts that is obtained by projecting each tuple $t \in \phi_1^D$ over attributes $\bar{K}$ and $\bar{V}$. More formally, for each tuple t of sort $(\bar{K}, \bar{V})$ it is the case that

$$\#(\phi^D, t) \ = \sum_{\{t' \in \phi_1^D \mid t'(\bar{K},\bar{V}) = \bar{t}\}} \#(\phi_1^D, t').$$

- If $\phi(\bar{K}, \bar{V}) = \mathsf{Group}_{\oplus}\phi_1(\bar{K}, \bar{V})$, then $\phi^D = \mathsf{Solve}_{\oplus}(\phi_1^D)$.

4. Assumptions for the expressive completeness result

As mentioned earlier, we want to use RA with aggregation as a yardstick for understanding the expressive power of Lara. However, there are important mismatches between the semantics of both languages that we need to solve before proceeding further. First, the evaluation of an $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ formula does not need to be an associative table, which is a property that Lara expressions enjoy by design. Second, the definition of Lara is very general and does not necessarily include some basic extension functions that are required for capturing standard operations expressible in relational algebra. We tackle these issues in this section, by first defining a modified syntax for expressions in $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ that ensures they always evaluate to an associative table, and, then, providing a minimal list of basic extension functions that suffice to capture relational algebra; namely, projecting over value-attributes, copying and renaming attributes, and selecting rows based on (in)equality.

4.1. Semantics based on associative tables

We define a syntactic restriction on the class of $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ expressions that ensures that the evaluation of any such an expression on a Lara database is an associative table. Notice that there are only two operations of $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ that do not satisfy this property. These two operations are the sum and the projection, as exemplified next.

Example 2. Suppose that we have two relations R and S of sort (K_1, K_2, V) and a Lara database D over this schema such that

$$R^D \ = \ S^D \ = \ \{(k_1, k, v), (k_2, k, v')\}.$$

Then $(R + S)^D = \{(k_1, k, v), (k_1, k, v), (k_2, k, v'), (k_2, k, v')\}$, which is not an associative table. Analogously, $(\pi_{K_2,V} R)^D = \{(k, v), (k, v')\}$, which is neither an associative table. □

To overcome this difficulty, we define a syntactic fragment of $\mathrm{RA}_{\mathsf{Agg}}(\Psi)$ which we call the class of *constrained expressions*. These expressions have the property that their evaluation over a Lara database D always yields an associative table. The way we achieve this is by restricting sums and projection to always appear under the scope of a grouping operation. In this way we ensure that the final result satisfies the "key-functionality" constraint of associative tables by definition.

Definition 4 *(Constrained expressions).* We denote by $\mathrm{RA}^c_{\mathsf{Agg}}(\Psi)$ the set of *constrained expressions* over σ, which is defined as follows:

- Expressions $\perp$ and R, for $R \in \sigma \cup \Psi$, are *atomic* constrained expressions.
- Constrained expressions are closed under renaming, join, selection, and difference.
- If $\phi_1(\bar{K}_1, \bar{V}_1), \phi_2(\bar{K}_2, \bar{V}_2)$ are constrained expressions, then

$$\mathsf{Group}_{\oplus}(\pi_{\bar{K},\bar{V}}\phi_1 + \pi_{\bar{K},\bar{V}}\phi_2)$$

is also a constrained expression, assuming that $\bar{K} \subseteq \bar{K}_1 \cap \bar{K}_2$ and $\bar{V} \subseteq \bar{V}_1 \cap \bar{V}_2$.

The following proposition asserts the self-evident fact that constrained expressions always yield associative tables when evaluated on Lara databases.

Proposition 2. *Let $\phi(\bar{K}, \bar{V})$ be a constrained expression in $\mathrm{RA}^c_{\mathsf{Agg}}(\Psi)$ and D a Lara database. Then ϕ^D is an associative table of sort $(\bar{K}, \bar{V})$.*

Before continuing, we make some observations regarding which derived expressions are also constrained.

- $\mathsf{Group}_\oplus(\phi_1 + \phi_2)$, for ϕ_1, ϕ_2 constrained expressions of the same sort $(\bar{K}', \bar{V}')$, is also a constrained expression. In fact, this expression is obtained as a particular case of $\mathsf{Group}_\oplus(\pi_{\bar{K},\bar{V}}\phi_1 + \pi_{\bar{K},\bar{V}}\phi_2)$ when $\bar{K}_1 = \bar{K}_2 = \bar{K} = \bar{K}'$ and $\bar{V}_1 = \bar{V}_2 = \bar{V} = \bar{V}'$.
- For any tuples $\bar{K}$ of key-attributes and $\bar{V}$ of value-attributes, there is a constrained expression $\bot(\bar{K}, \bar{V})$ of sort $(\bar{K}, \bar{V})$ that is interpreted as the empty relation over any database D. In fact, take any relation symbol $R(\bar{K}, \bar{V})$ in σ. By definition, $\bar{K}$ and $\bar{V}$ are nonempty, and hence there is a key-attribute $K \in \bar{K}$ and a value-attribute $V \in \bar{V}$. With $\pi_{K,V}(R \bowtie \bot)$ we obtain then the expression $\bot(K, V)$. Now $\bot(\bar{K}, \bar{V})$ can be obtained by taking the cartesian product of sufficiently many suitable renamings of this expression.
- $\mathsf{Group}_\oplus(\pi_{\bar{K},\bar{V}}\phi_1)$, for ϕ_1 of sort $(\bar{K}_1, \bar{V}_1)$. Again, this expression is obtained as a particular case of $\mathsf{Group}_\oplus(\pi_{\bar{K},\bar{V}}\phi_1 + \pi_{\bar{K},\bar{V}}\phi_2)$ when $\phi_2 = \bot(\bar{K}, \bar{V})$.
- $\pi_{\bar{K},\bar{V}}\phi_1$, for ϕ_1 a constrained expression of sort $(\bar{K}, \bar{V}')$, is a constrained expression. In fact, this expression can be obtained from $\mathsf{Group}_\oplus(\pi_{\bar{K},\bar{V}}\phi_1)$ by setting $\oplus$ to be an aggregate operator that acts as the identity on any singleton. The reason is that when ϕ_1 is of sort $(\bar{K}, \bar{V}')$, then $\pi_{\bar{K},\bar{V}}\phi_1$ does not remove any key attribute from the evaluation e_1^D of e_1 on a Lara database D, which implies that $(\pi_{\bar{K},\bar{V}}\phi_1)^D$ is an associative table.

In our proofs we thus assume that these expressions are in fact constrained expressions from $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi)$.

4.2. Availability of extension functions

As explained before, we require some natural assumptions on the extension functions that Lara is allowed to use. In particular, we need these functions to be able to express traditional relational algebra operations that are not included in the core of Lara; namely, projecting over value-attributes, renaming attributes, and selecting rows based on (in)equality. We then assume that Lara(Ω) contains the following families of extension functions and derived expressions. (In the following we abuse notation and write $f(e)$, for f an extension function and e a Lara expression, instead of $\mathsf{Ext}_f\, e$.)

Projection on values: We assume that Ω contains all extension functions of the form $\Pi^{\bar{K},\bar{V}}_{\mathsf{Values}}$, which take as input a tuple $(\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V}_1)$, for $\bar{V} \subseteq \bar{V}_1$, and output the tuple $(\emptyset, \bar{v}')$ of sort $(\emptyset, \bar{V}')$ such that $\bar{v} = \bar{v}'{\downarrow_{\bar{V}}}$. That is, the evaluation of $\Pi^{\bar{K},\bar{V}}_{\mathsf{Values}}(e)$ is obtained from the evaluation of e by discarding the attributes in $\bar{V}_1 \setminus \bar{V}$. Notice that $\Pi^{\bar{K},\bar{V}}_{\mathsf{Values}}(e)$ is of sort $(\bar{K}, \bar{V}')$ by definition, but we can rename back $\bar{V}'$ into $\bar{V}$ by performing another projection over $(\bar{K}, \bar{V}')$. By slightly abusing terminology, we assume then that the result of $\Pi^{\bar{K},\bar{V}}_{\mathsf{Values}}(e)$ is of sort $(\bar{K}, \bar{V})$.

Renaming attributes: All extension functions of the form $\mathsf{copy}_{\bar{K},\bar{K}'}$ are in Ω, where $\bar{K}, \bar{K}'$ are tuples of key-attributes of the same arity. The function $\mathsf{copy}_{\bar{K},\bar{K}'}$ takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}_1, \bar{V})$, for $\bar{K} \subseteq \bar{K}_1$, and produces a tuple $t' = (\bar{k}', \bar{v})$ of sort $(\bar{K}', \bar{V}')$ such that $t'(\bar{K}') = t(\bar{K})$, i.e., $\mathsf{copy}_{\bar{K},\bar{K}'}$ copies the value of attributes $\bar{K}$ in the new attributes $\bar{K}'$. Notice that $\mathsf{copy}_{\bar{K},\bar{K}'}(e)$ is of sort $(\bar{K}_1, \bar{K}', \bar{V}')$ by definition, but as before we can rename back $\bar{V}'$ into $\bar{V}$ by performing a projection over $(\bar{K}, \bar{K}_1, \bar{V}')$. By slightly abusing terminology, we can assume then that the result of $\mathsf{copy}_{\bar{K},\bar{K}'}(e)$ is of sort $(\bar{K}_1, \bar{K}', \bar{V})$.

With this extension function we can also define the *renaming* expression

$$\mathsf{rename}_{\bar{K}\to\bar{K}'}\, e \;:=\; \Pi^{\bar{K}}\; (\mathsf{copy}_{\bar{K},\bar{K}'}(e)),$$

where Π has no subscript $\oplus$ as no aggregation is necessary in this case. Recall that if e is of sort $(\bar{K}_1, \bar{V})$, for $\bar{K}, \bar{K}' \subseteq \bar{K}_1$, then $\Pi^{\bar{K}}\, e$ computes the projection of e over all attributes except for $\bar{K}$, and hence $\mathsf{rename}_{\bar{K}\to\bar{K}'}\, e$ simply renames the attributes in $\bar{K}$ to $\bar{K}'$ over the evaluation of e.

Also, Ω contains all extension functions $\mathsf{copy}_{\bar{V},\bar{V}'}$, where $\bar{V}, \bar{V}'$ are tuples of value-attributes of the same arity. This extension function takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V}_1)$, where $\bar{V} \subseteq \bar{V}_1$, and produces a tuple $t' = (\emptyset, \bar{v}')$ of sort $(\emptyset, \bar{V}_1', \bar{V}')$ such that $t'(\bar{V}_1') = t(\bar{V}_1)$ and $t'(\bar{V}') = t(\bar{V})$, i.e., $\mathsf{copy}_{\bar{V},\bar{V}'}$ copies the attributes in $\bar{V}$ to $\bar{V}'$. Although $\mathsf{copy}_{\bar{V},\bar{V}'}(e)$ is of sort $(\bar{K}, \bar{V}_1', \bar{V}')$ by definition, we can turn it into one an equivalent one of sort $(\bar{K}, \bar{V}_1, \bar{V}')$ by performing a projection on value-attributes. Thus, we can safely assume that $\mathsf{copy}_{\bar{V},\bar{V}'}(e)$ is of sort $(\bar{K}, \bar{V}_1, \bar{V}')$.

With this extension function we can also define the *renaming* expression

$$\mathsf{rename}_{\bar{V}\to\bar{V}'}\, e \;:=\; \Pi^{\bar{K},\bar{V}_1\setminus\bar{V},\bar{V}'}_{\mathsf{Values}}\; (\mathsf{copy}_{\bar{V},\bar{V}'}(e)).$$

It is easy to see that $\mathsf{rename}_{\bar{V}\to\bar{V}'}\, e$ simply renames the attributes in $\bar{V}$ to $\bar{V}'$ over the evaluation of e.

Filtering: All extension functions of the form $\mathsf{eq}_{\bar{K},\bar{K}'}$ and $\mathsf{neq}_{\bar{K},\bar{K}'}$, where $\bar{K}, \bar{K}'$ are tuples of key-attributes of the same arity, are in Ω. The function $\mathsf{eq}_{\bar{K},\bar{K}'}$ takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}_1, \bar{V})$, for $\bar{K}, \bar{K}' \subseteq \bar{K}_1$, and produces as output

Table $\phi^D_{e_1}$

K	K'	V	V'
k_1	k'_1	3	5
k_2	k'_2	1	3
k_3	k'_3	2	4
k_3	k''_3	0	1

Table $\phi^D_{e_2}$

K	K''	V	V''
k_1	k''_1	3	7
k_2	k''_2	2	0
k_5	k''_5	7	0

Fig. 5. The tables e^D_1 and e^D_2 from the proof of Theorem 1.

the tuple $t' = (\emptyset, \bar{v})$ of sort $(\emptyset, \bar{V}')$, if $t(\bar{K}) = t(\bar{K}')$, and the empty associative table otherwise. Hence, this function acts as a filter over an associative table of sort $(\bar{K}_1, \bar{V})$, keeping only those tuples t such that $t(\bar{K}) = t(\bar{K}')$. The definition of extension functions $\mathsf{neq}_{\bar{K},\bar{K}'}$ is exactly the same, only that we now keep only those tuples t such that $t(\bar{K}) \neq t(\bar{K}')$. Although the evaluation of these functions is of sort $(\bar{K}_1, \bar{V}')$ by definition, we can make it of sort $(\bar{K}_1, \bar{V})$ by performing a projection on value-attributes. Thus, we can safely assume that $\mathsf{eq}_{\bar{K},\bar{K}'}(e)$ and $\mathsf{neq}_{\bar{K},\bar{K}'}(e)$ are of sort $(\bar{K}_1, \bar{V})$.

Analogously, we have extension functions $\mathsf{eq}_{\bar{V},\bar{V}'}$ and $\mathsf{neq}_{\bar{V},\bar{V}'}$, where $\bar{V}, \bar{V}'$ are tuples of key-attributes of the same arity. The function $\mathsf{eq}_{\bar{V},\bar{V}'}$ takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V}_1)$, for $\bar{V}, \bar{V}' \subseteq \bar{V}_1$, and produces as output the tuple $t' = (\emptyset, \bar{v})$ of sort $(\emptyset, \bar{V}'_1)$, if $t(\bar{V}) = t(\bar{V}')$, and the empty associative table otherwise. Hence, this function acts as a filter over an associative table of sort $(\bar{K}, \bar{V}_1)$, keeping only those tuples t such that $t(\bar{V}) = t(\bar{V}')$. The definition of extension functions $\mathsf{neq}_{\bar{V},\bar{V}'}$ is exactly the same, only that we now keep only those tuples t such that $t(\bar{V}) \neq t(\bar{V}')$. As before, we can safely assume that $\mathsf{eq}_{\bar{V},\bar{V}'}(e)$ and $\mathsf{neq}_{\bar{V},\bar{V}'}(e)$ are of sort $(\bar{K}, \bar{V}_1)$.

In addition, for each aggregate operator $\oplus$ and tuple $\bar{V}$ of value-attributes, we have that Ω contains extension functions $\mathsf{eq}_{\bar{V}=(0_\oplus,\dots,0_\oplus)}$ and $\mathsf{neq}_{\bar{V}=(0_\oplus,\dots,0_\oplus)}$. The first one takes as input a tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V}_1)$, for $\bar{V} \subseteq \bar{V}_1$, and produces as output the tuple $t = (\emptyset, \bar{v})$ of sort $(\emptyset, \bar{V}'_1)$, if $t(\bar{V}) = (0_\oplus, \dots, 0_\oplus)$, and the empty associative table otherwise. Hence, this function acts as a filter over an associative table of sort $(\bar{K}, \bar{V}_1)$, keeping only those tuples t for which $t(\bar{V}) = (0_\oplus, \dots, 0_\oplus)$. In the same way we define $\mathsf{neq}_{\bar{V}=(0_\oplus,\dots,0_\oplus)}$, which in this case retains only those tuples t for which $t(\bar{V}) \neq (0_\oplus, \dots, 0_\oplus)$. As before, we can safely assume that $\mathsf{eq}_{\bar{V}=(0_\oplus,\dots,0_\oplus)}(e)$ and $\mathsf{neq}_{\bar{V}=(0_\oplus,\dots,0_\oplus)}(e)$ are of sort $(\bar{K}, \bar{V}_1)$.

5. Expressive completeness of Lara with respect to $\mathsf{RA}_{\mathsf{Agg}}$

We prove that Lara(Ω) has the same expressive power as the constrained fragment of $\mathsf{RA}_{\mathsf{Agg}}(\Psi_\Omega)$, where Ψ_Ω is a set that contains relations that represent the extension functions in Ω, assuming that Ω contains all distinguished extension functions defined in the previous section. By definition, for every extension function $f \in \Omega$ of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, there is a relation $R_f \subseteq \mathsf{Keys}^{|\bar{K}|+|\bar{K}'|} \times \mathsf{Values}^{|\bar{V}|+|\bar{V}'|}$ in Ψ_Ω such that

$$R_f \ = \{(\bar{k}, \bar{k}', \bar{v}, \bar{v}') \mid (\bar{k}, \bar{v}) \in \mathsf{Keys}^{|\bar{K}|} \times \mathsf{Values}^{|\bar{V}|} \text{ and } (\bar{k}', \bar{v}') \in f(\bar{k}, \bar{v})\}.$$

5.1. From LARA to RA with aggregation

We show first that the expressive power of Lara(Ω) is bounded by that of $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$.

Theorem 1. *For every expression $e[\bar{K}, \bar{V}]$ of* Lara(Ω) *there is a constrained expression $\phi_e(\bar{K}, \bar{V})$ of $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$ with $e^D = \phi^D_e$, for each* Lara *database D.*

Proof of Theorem 1. The proof is by induction on e. We start with the base cases. If $e = \emptyset$, then $\phi_e = \bot$. If $e = R[\bar{K}, \bar{V}]$, for $R \in \sigma$, then $\phi_e(\bar{K}, \bar{V}) = R(\bar{K}, \bar{V})$. We now handle the inductive cases. We use a running example to illustrate the main ideas. We assume that we have two constrained expressions $\phi_{e_1}(K, K', V, V')$ and $\phi_{e_2}(K, K'', V, V'')$ of $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$, and a Lara database D, and it holds that

$$\phi^D_{e_1} = \{(k_1, k'_1, 3, 5), (k_2, k'_2, 1, 3), (k_3, k'_3, 2, 4), (k_3, k''_3, 0, 1)\} \text{ and}$$
$$\phi^D_{e_2} = \{(k_1, k''_1, 3, 7), (k_2, k''_2, 2, 0), (k_5, k''_5, 7, 0))\}.$$

This is shown in Fig. 5 in the form of associative tables.

Join. Consider the expression $e[\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = e_1[\bar{K}_1, \bar{V}_1] \bowtie_\otimes e_2[\bar{K}_2, \bar{V}_2]$, and assume that $\phi_{e_1}(\bar{K}_1, \bar{V}_1)$ and $\phi_{e_2}(\bar{K}_2, \bar{V}_2)$ are the constrained expressions in $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$ obtained for $e_1[\bar{K}_1, \bar{V}_1]$ and $e_2[\bar{K}_2, \bar{V}_2]$, respectively, by induction hypothesis. We define the following auxiliary constrained expressions in $\mathsf{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$:

- $\alpha_1(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \setminus \bar{V}_2) := (\pi_{\bar{K}_1, \bar{V}_1 \setminus \bar{V}_2} \phi_{e_1}) \bowtie (\pi_{\bar{K}_2} \phi_{e_2})$, which computes over a LARA database D all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1{\downarrow}_{\bar{V}_1 \setminus \bar{V}_2})$, such that there are $(\bar{k}_1, \bar{v}_1) \in \phi^D_{e_1}$ and $(\bar{k}_2, \bar{v}_2) \in \phi^D_{e_2}$ for which $\bar{k}_1$ and $\bar{k}_2$ are compatible with respect to $\bar{K}_1 \cap \bar{K}_2$. Following our running example, the result of evaluating α_1 on D is the set $\{(k_1, k'_1, k''_1, 5), (k_2, k'_2, k''_2, 3)\}$.
- $\alpha_2(\bar{K}_1 \cup \bar{K}_2, \bar{V}_2 \setminus \bar{V}_1) := (\pi_{\bar{K}_1} \phi_{e_1}) \bowtie (\pi_{\bar{K}_2, \bar{V}_2 \setminus \bar{V}_1} \phi_{e_2})$, which computes over a LARA database D all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_2{\downarrow}_{\bar{V}_2 \setminus \bar{V}_1})$, such that there are $(\bar{k}_1, \bar{v}_1) \in \phi^D_{e_1}$ and $(\bar{k}_2, \bar{v}_2) \in \phi^D_{e_2}$ for which $\bar{k}_1$ and $\bar{k}_2$ are compatible with respect to $\bar{K}_1 \cap \bar{K}_2$. Following our running example, the result of evaluating α_2 on D is the set $\{(k_1, k'_1, k''_1, 7), (k_2, k'_2, k''_2, 0)\}$.
- $\beta_1(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cap \bar{V}_2) := (\pi_{\bar{K}_1, \bar{V}_1 \cap \bar{V}_2} \phi_{e_1}) \bowtie (\pi_{\bar{K}_2} \phi_{e_2})$, which computes over a LARA database D all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1{\downarrow}_{\bar{V}_1 \cap \bar{V}_2})$, such that there are $(\bar{k}_1, \bar{v}_1) \in \phi^D_{e_1}$ and $(\bar{k}_2, \bar{v}_2) \in \phi^D_{e_2}$ for which $\bar{k}_1$ and $\bar{k}_2$ are compatible with respect to $\bar{K}_1 \cap \bar{K}_2$. Following our running example, the result of evaluating β_1 on D is the set $\{(k_1, k'_1, k''_1, 3), (k_2, k'_2, k''_2, 1)\}$.
- $\beta_2(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cap \bar{V}_2) := (\pi_{\bar{K}_1} \phi_{e_1}) \bowtie (\pi_{\bar{K}_2, \bar{V}_1 \cap \bar{V}_2} \phi_{e_2})$, which computes over a LARA database D all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_2{\downarrow}_{\bar{V}_1 \cap \bar{V}_2})$, such that there are $(\bar{k}_1, \bar{v}_1) \in \phi^D_{e_1}$ and $(\bar{k}_2, \bar{v}_2) \in \phi^D_{e_2}$ for which $\bar{k}_1$ and $\bar{k}_2$ are compatible with respect to $\bar{K}_1 \cap \bar{K}_2$. Following our running example, the result of evaluating β_2 on D is the set $\{(k_1, k'_1, k''_1, 3), (k_2, k'_2, k''_2, 2)\}$.

Next, we define a constrained expression

$$\beta(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cap \bar{V}_2) = \mathsf{Group}_\otimes(\beta_1 + \beta_2).$$

Over a LARA database D, this expression computes all tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{u})$ such that $(\bar{k}_1 \cup \bar{k}_2, \bar{u}_1) \in \beta_1^D$, $(\bar{k}_1 \cup \bar{k}_2, \bar{u}_2) \in \beta_2^D$, and $\bar{u}_1 \otimes \bar{u}_2 = \bar{u}$. Notice that the use of the multiset semantics for $\mathrm{RA_{Agg}}$ is essential for this step, as otherwise when $\bar{u}_1 = \bar{u}_2$ we would have that $\beta_1 + \beta_2$ reduces to $\beta_1 \cup \beta_2$, and the latter only contains one tuple of the form $(\bar{k}_1 \cup \bar{k}_2, \cdot)$, namely, $(\bar{k}_1 \cup \bar{k}_2, \bar{u}_1)$. The evaluation of $\mathsf{Group}^{\bar{K}_1 \cup \bar{K}_2}_\otimes(\beta_1 + \beta_2)$ might not necessarily yield the tuple $(\bar{k}_1 \cup \bar{k}_2, \bar{u}_1 \otimes \bar{u}_1)$ then, which is what the semantics of LARA demands.

Following with our running example, the result of evaluating β on D, with respect to the aggregate operator $\otimes$ of multiplication on $\mathbb{N}$, is the set

$$\mathsf{Group}_\otimes \{\!\{(k_1, k'_1, k''_1, 3), (k_1, k'_1, k''_1, 3), (k_2, k'_2, k''_2, 1), (k_2, k'_2, k''_2, 2)\}\!\} = \{(k_1, k'_1, k''_1, 9), (k_2, k'_2, k''_2, 2)\}.$$

It should be clear, then, that

$$\phi_e := (\alpha_1 \bowtie \alpha_2) \bowtie \beta.$$

In fact, by definition and inductive hypothesis we have that the evaluation of $e_1[\bar{K}_1, \bar{V}_1] \bowtie_\otimes e_2[\bar{K}_2, \bar{V}_2]$ on a LARA database D contains the tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v})$ such that $(\bar{k}_1, \bar{v}_1) \in e^D_{\phi_1}$, $(\bar{k}_2, \bar{v}_2) \in e^D_{\phi_2}$, the tuples $\bar{k}_1$ and $\bar{k}_2$ are compatible, and $\bar{v} = \mathrm{pad}^{\bar{V}_2}_\otimes(\bar{v}_1) \otimes \mathrm{pad}^{\bar{V}_1}_\otimes(\bar{v}_2)$. But then, by definition,

$$\bar{v} = \big(\bar{v}_1{\downarrow}_{\bar{V}_1 \setminus \bar{V}_2}, (\bar{v}_1{\downarrow}_{\bar{V}_1 \cap \bar{V}_2} \otimes \bar{v}_2{\downarrow}_{\bar{V}_1 \cap \bar{V}_2}), \bar{v}_2{\downarrow}_{\bar{V}_2 \setminus \bar{V}_1}\big).$$

The tuples of the form $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1{\downarrow}_{\bar{V}_1 \setminus \bar{V}_2})$ are obtained as the result of α_1; those of the form $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_2{\downarrow}_{\bar{V}_1 \setminus \bar{V}_2})$ as the result of α_2; and those of the form $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1{\downarrow}_{\bar{V}_1 \cap \bar{V}_2} \otimes \bar{v}_2{\downarrow}_{\bar{V}_1 \cap \bar{V}_2})$ as the result of β. The join builds the final result.

Following with our running example, the result of evaluating ϕ_e on D is the set

$$\{(k_1, k'_1, k''_1, 5, 9, 7), (k_2, k'_2, k''_2, 3, 2, 0)\}.$$

This is, as expected, equivalent to the evaluation of e on D.

Union. Consider the expression $e[\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2] = e_1[\bar{K}_1, \bar{V}_1] \,\bar{\bowtie}_\oplus\, e_2[\bar{K}_2, \bar{V}_2]$, and assume that $\phi_{e_1}(\bar{K}_1, \bar{V}_1)$ and $\phi_{e_2}(\bar{K}_2, \bar{V}_2)$ are the constrained expressions in $\mathrm{RA}^c_{\mathrm{Agg}}(\Psi_\Omega)$ obtained for $e_1[\bar{K}_1, \bar{V}_1]$ and $e_2[\bar{K}_2, \bar{V}_2]$, respectively, by induction hypothesis.

Before giving the translation, we explain how it is possible to construct in $\mathrm{RA}^c_{\mathrm{Agg}}(\Psi_\Omega)$ an expression $\mathsf{Neutral}_\oplus(\bar{V})$, where $\bar{V}$ is an arbitrary set of value-attributes, whose interpretation over D is the single tuple $(0_\oplus, \dots, 0_\oplus)$ of sort $\bar{V}$. Take an arbitrary relation symbol $R \in \sigma$ and suppose that its sort is $(\bar{K}, \bar{V}')$. By definition of schema, $\bar{V}'$ is nonempty, and by definition of database, $R^D \neq \emptyset$. Suppose that V is some attribute in $\bar{V}'$. We define a constrained expression $\mathsf{Neutral}_\oplus(V) := \mathsf{Group}_{\mathsf{Zero}(\oplus)}(\pi_V R)$, where $\mathsf{Zero}(\oplus)$ denotes the aggregate operator that on any nonempty multiset of elements from Values it returns the neutral value $0_\oplus$ for $\oplus$. Then the evaluation of $\mathsf{Neutral}_\oplus(V)$ on D consists of the single tuple $(0_\oplus)$ of sort V. It is easy to see that, by performing this operation $|\bar{V}|$ times and taking the join of suitable renamings of the obtained expressions, we obtain an expression $\mathsf{Neutral}_\oplus(\bar{V})$ whose interpretation over D is the tuple $(0_\oplus, \dots, 0_\oplus)$ of sort $\bar{V}$.

We now explain how to define ϕ_e. We make use of the auxiliary constrained expressions defined below:

- $\alpha_1(\bar{K}_1, \bar{V}_1 \cup \bar{V}_2) := \phi_{e_1}(\bar{K}_1, \bar{V}_1) \bowtie \mathsf{Neutral}_\oplus(\bar{V}_2 \setminus \bar{V}_1)$. Notice that, by definition, α_1 computes over a LARA database D the set of tuples $(\bar{k}_1, \mathrm{pad}^{\bar{V}_2}_\oplus(\bar{v}_1))$, for $(\bar{k}_1, \bar{v}_1) \in \phi^D_{e_1}$. For our running example, assuming that $\oplus$ is the sum operation on $\mathbb{N}$, the result of evaluating α_1 on D is the set of tuples of sort (K, K', V, V', V''):

$$\{(k_1, k_1', 3, 5, 0), (k_2, k_2', 1, 3, 0), (k_3, k_3', 2, 4, 0), (k_3, k_3'', 0, 1, 0)\}.$$

- $\alpha_2(\bar{K}_2, \bar{V}_1 \cup \bar{V}_2) := \phi_{e_2}(\bar{K}_2, \bar{V}_2) \bowtie \mathsf{Neutral}_{\oplus}(\bar{V}_1 \setminus \bar{V}_2)$. Notice that, by definition, α_2 computes over a Lara database D the set of tuples $(\bar{k}_2, \mathrm{pad}_{\oplus}^{\bar{V}_1}(\bar{v}_2))$, for $(\bar{k}_2, \bar{v}_2) \in \phi_{e_2}^D$. For our running example, assuming that $\oplus$ is the sum operation on $\mathbb{N}$, the result of evaluating α_2 on D is the set of sort (K, K'', V, V', V''):

$$\{(k_1, k_1'', 3, 0, 7), (k_2, k_2'', 2, 0, 0), (k_5, k_5'', 7, 0, 0)\}.$$

It should be clear, then, that ϕ_e can be defined as the constrained expression

$$\mathsf{Group}_{\oplus}\left(\pi_{\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2} \alpha_1 + \pi_{\bar{K}_1 \cap \bar{K}_2, \bar{V}_1 \cup \bar{V}_2} \alpha_2\right).$$

In fact, by definition we have that when ϕ_e is evaluated on a Lara database D it yields the set

$$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}) \in \alpha_1^D \text{ and } \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}, \text{ or } (\bar{k}_2, \bar{v}) \in \alpha_2^D \text{ and } \bar{k} = \bar{k}_2{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}\}\!\},$$

which is equivalent to the set

$$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in \phi_{e_1}^D, \ \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}, \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\bar{v}_1), \text{ or } (\bar{k}_2, \bar{v}_2) \in \phi_{e_2}^D, \ \bar{k} = \bar{k}_2{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}, \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\bar{v}_2)\}\!\}.$$

By induction hypothesis, this is equivalent to

$$\mathsf{Solve}_{\oplus}\{\!\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in e_1^D, \ \bar{k} = \bar{k}_1{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}, \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_2}(\bar{v}_1), \text{ or } (\bar{k}_2, \bar{v}_2) \in e_2^D, \ \bar{k} = \bar{k}_2{\downarrow}_{\bar{K}_1 \cap \bar{K}_2}, \text{ and } \bar{v} = \mathrm{pad}_{\oplus}^{\bar{V}_1}(\bar{v}_2)\}\!\},$$

which corresponds to the evaluation of $e = e_1 \between_{\oplus} e_2$ on D by definition.

For our running example, assuming that $\oplus$ is the sum operation on $\mathbb{N}$, the result of evaluating ϕ_e on D is the set of sort (K, V, V', V''):

$$\mathsf{Group}_{\otimes}\{\!\{(k_1, 3, 5, 0), (k_2, 1, 3, 0), (k_3, 2, 4, 0), (k_3, 0, 1, 0), (k_1, 3, 0, 7), (k_2, 2, 0, 0), (k_5, 7, 0, 0)\}\!\} = \{(k_1, 6, 5, 7), (k_2, 3, 3, 0), (k_3, 2, 5, 0), (k_5, 7, 0, 0)\}.$$

As expected, this coincides with the evaluation of e on D.

Extend. Consider the expression $e[\bar{K} \cup \bar{K}', \bar{V}'] = \mathsf{Ext}_f\, e_1[\bar{K}, \bar{V}]$, where f is of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$, and assume that $\phi_{e_1}(\bar{K}, \bar{V})$ is the constrained expression obtained for $e_1[\bar{K}, \bar{V}]$ by induction hypothesis. Recall that by definition of extension function we have $\bar{K} \cap \bar{K}' = \emptyset$ and $\bar{V} \cap \bar{V}' = \emptyset$. It should be clear then that $\phi_e(\bar{K}, \bar{K}', \bar{V}') := \pi_{\bar{K}, \bar{K}', \bar{V}'}(\phi_{e_1}(\bar{K}, \bar{V}) \bowtie R_f(\bar{K}, \bar{K}', \bar{V}, \bar{V}'))$. □

The translation from Lara to $\mathrm{RA}_{\mathsf{Agg}}$ given in the proof of Theorem 1 does not require the use of the difference operator. This is in line with our results in the next section, where we show that this operator can be encoded in Lara by a suitable combination of aggregate operators and extension functions.

5.2. From RA with aggregation to LARA

We now prove the other direction of the expressive completeness result.

Theorem 2. *For each constrained expression $\phi(\bar{K}, \bar{V})$ of $\mathrm{RA}^c_{\mathsf{Agg}}(\Psi_\Omega)$, there is a Lara($\Omega$) expression $e_\phi[\bar{K}, \bar{V}]$ with $e_\phi^D = \phi^D$, for each Lara database D.*

Proof of Theorem 2. We prove the theorem by induction on ϕ. We start with the base cases. If $\phi = \bot$, then $e_\phi = \emptyset$. If $\phi = R(\bar{K}, \bar{V})$, for $R \in \sigma$, then $e_\phi(\bar{K}, \bar{V}) = R[\bar{K}, \bar{V}]$. We now consider the inductive cases. We use a running example to illustrate the main ideas. We assume that we have two Lara(Ω) expressions $e_{\phi_1}[K, V]$ and $e_{\phi_2}[K, V]$ and a Lara database D, and it holds that

$$e_{\phi_1}^D = \{(k_1, 0), (k_2, 1), (k_3, 2), (k_4, 0)\} \text{ and } e_{\phi_2}^D = \{(k_1, 0), (k_2, 2), (k_5, 7))\}.$$

Renaming. Assume that $\phi = \rho_{\bar{K}\to\bar{K}',\bar{V}\to\bar{V}'}(\phi_1(\bar{K}_1, \bar{V}_1))$. Then

$$e_\phi \;:=\; \text{rename}_{\bar{K}\to\bar{K}'}\text{rename}_{\bar{V}\to\bar{V}'}(e_{\phi_1}[\bar{K}_1, \bar{V}_1]),$$

where e_{ϕ_1} is the LARA(Ω) expression obtained for ϕ_1 by induction hypothesis.

Join. Assume that $\phi = \phi_1(\bar{K}_1, \bar{V}_1) \bowtie \phi_2(\bar{K}_2, \bar{V}_2)$, where ϕ_1, ϕ_2 are constrained expressions. Let $e_{\phi_1}[\bar{K}_1, \bar{V}_1]$ and $e_{\phi_2}[\bar{K}_2, \bar{V}_2]$ be the LARA(Ω) expressions obtained for ϕ_1 and ϕ_2, respectively, by induction hypothesis. By definition, the evaluation of ϕ on a LARA database D is the set that contains the tuples $(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \cup \bar{v}_2)$ such that $(\bar{k}_1, \bar{v}_1) \in e^D_{\phi_1}$, $(\bar{k}_2, \bar{v}_2) \in e^D_{\phi_2}$, the tuples $\bar{k}_1$ and $\bar{k}_2$ are compatible over $\bar{K}_1 \cap \bar{K}_2$, and so are $\bar{v}_1$ and $\bar{v}_2$ over $\bar{V}_1 \cap \bar{V}_2$. It is not hard to see that we can then define ϕ in LARA with the following expression:

$$e_\phi \;:= \textstyle\bigvee\!\!\!\!\bigwedge_{\text{Values}}^{\bar{K}_1\cup\bar{K}_2,\bar{V}_1,\bar{V}_2\setminus\bar{V}_1} \left[\text{eq}_{\bar{V}_1\cap\bar{V}_2,\bar{V}'}\big(e_{\phi_1} \bowtie (\text{rename}_{\bar{V}_1\cap\bar{V}_2\to\bar{V}'}(e_{\phi_2}))\big)\right],$$

where there is no need to specify an aggregate operator for the join $\bowtie$ as the two expressions involved share no value-attributes. In fact, the expression $e_{\phi_1} \bowtie (\text{rename}_{\bar{V}_1\cap\bar{V}_2\to\bar{V}'}(e_{\phi_2}))$, which is of sort $(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1, \bar{V}_2 \setminus \bar{V}_1, \bar{V}')$, computes the set of tuples

$$(\bar{k}_1 \cup \bar{k}_2, \bar{v}_1, \bar{v}_2{\downarrow}_{\bar{V}_2\setminus\bar{V}_1}, \bar{v}_2{\downarrow}_{\bar{V}_1\cap\bar{V}_2})$$

such that $(\bar{k}_1, \bar{v}_1) \in \phi^D_1$, $(\bar{k}_2, \bar{v}_2) \in \phi^D_2$, and the tuples $\bar{k}_1$ and $\bar{k}_2$ are compatible over $\bar{K}_1 \cap \bar{K}_2$. The application of $\text{eq}_{\bar{V}_1\cap\bar{V}_2,\bar{V}'}$ on top of this expression filters precisely those tuples of the above form that satisfy $\bar{v}_2{\downarrow}_{\bar{V}_1\cap\bar{V}_2} = \bar{v}_1{\downarrow}_{\bar{V}_1\cap\bar{V}_2}$, i.e., $\bar{v}_1$ and $\bar{v}_2$ are compatible over $\bar{V}_1 \cap \bar{V}_2$. By discarding the attributes in $\bar{V}'$ with $\bigvee\!\!\!\!\bigwedge_{\text{Values}}^{\bar{K}_1\cup\bar{K}_2,\bar{V}_1,\bar{V}_2\setminus\bar{V}_1}$ we then obtain the desired result.

Following our running example, we have that the evaluation of

$$e_{\phi_1} \bowtie (\text{rename}_{V\to V'}(e_{\phi_2}))$$

is the set $\{(k_1, 0, 0), (k_2, 1, 2)\}$, and hence the evaluation of

$$\text{eq}_{V,V'}\big(e_{\phi_1} \bowtie (\text{rename}_{\bar{V}\to V'}(e_{\phi_2}))$$

is $\{(k_1, 0, 0)\}$. To finalize, $e^D_\phi = \{(k_1, 0)\}$, which is equivalent to ϕ^D.

Selection. Assume that ϕ is of the form $\sigma_{K_1=K_2}\phi'(\bar{K}, \bar{V})$, where $K_1, K_2 \in \bar{K}$. All other cases are analogous, so we only handle this one. We can then define ϕ in LARA with the following expression:

$$e_\phi \;=\; \text{eq}_{K_1,K_2}(e_{\phi'}),$$

where $e_{\phi'}[\bar{K}, \bar{V}]$ is the LARA(Ω) expression obtained for ϕ' by induction hypothesis.

Difference. This is an interesting case, as LARA does not include any kind of complementation feature explicitly. Nevertheless, and as we show below, difference over associative tables can be encoded in LARA by using the appropriate extension functions.

Assume that $\phi = \phi_1(\bar{K}, \bar{V}) \setminus \phi_2(\bar{K}, \bar{V})$, where ϕ_1, ϕ_2 are constrained expressions. Let $e_{\phi_1}[\bar{K}, \bar{V}]$ and $e_{\phi_2}[\bar{K}, \bar{V}]$ be the LARA(Ω) expressions obtained for ϕ_1 and ϕ_2, respectively, by induction hypothesis. Recall that e_{ϕ_1} and e_{ϕ_2} evaluate to associative tables by definition, and hence the evaluation ϕ^D of ϕ on a LARA database D is the set that contains (a) the tuples $(\bar{k}, \bar{v}) \in e^D_{\phi_1}$ such that there is a tuple of the form $(\bar{k}, \bar{w}) \in e^D_{\phi_2}$ for which $\bar{v} \neq \bar{w}$, and (b) the tuples $(\bar{k}, \bar{v}) \in e^D_{\phi_1}$ for which there is no tuple of the form $(\bar{k}, \bar{w}) \in e^D_{\phi_2}$.

Following our running example, we have that

$$\phi^D \;=\; e^D_{\phi_1} \setminus e^D_{\phi_2} \;=\; \{(k_2, 1), (k_3, 2), (k_4, 0)\}.$$

In fact, the tuple $(k_2, 1)$ appears in the result because even though $e^D_{\phi_2}$ contains a tuple of the form (k_2, v), it is the case that $v \neq 1$; the tuples $(k_3, 2)$ and $(k_4, 0)$ appear in the result because $e^D_{\phi_2}$ contains neither a tuple of the form (k_3, v) nor one of the form (k_4, v); and the tuple $(k_1, 0)$ does not appear in the result because it appears in both $e^D_{\phi_1}$ and $e^D_{\phi_2}$.

For the proof, we partition ϕ^D into three sets:

(P1) The tuples $(\bar{k}, \bar{v}) \in e^D_{\phi_1}$ such that there is a tuple of the form $(\bar{k}, \bar{w}) \in e^D_{\phi_2}$ for which $\bar{v} \neq \bar{w}$.
(P2) The tuples $(\bar{k}, \bar{v}) \in e^D_{\phi_1}$ such that (a) $\bar{v} \neq (0_\oplus, \ldots, 0_\oplus)$, and (b) there is no tuple of the form $(\bar{k}, \bar{v}') \in e^D_{\phi_2}$.
(P3) The tuples $(\bar{k}, 0_\oplus, \ldots, 0_\oplus) \in e^D_{\phi_1}$ such that there is no tuple of the form $(\bar{k}, \bar{v}) \in e^D_{\phi_2}$.

We show that each one of these sets can be expressed as an expression in LARA(Ω), and thus that ϕ itself can be expressed in LARA(Ω).

First, we take the LARA expression

$$e_\alpha[\bar{K},\bar{V}] := \bigvee^{\bar{K},\bar{V}}_{\mathsf{Values}}\left[\mathsf{neq}_{\bar{V},\bar{V}'}\big(e_{\phi_1} \bowtie \mathsf{rename}_{\bar{V}\to\bar{V}'}(e_{\phi_2})\big)\right],$$

where there is no need to specify an aggregate operator for $\bowtie$ as $\bar{V}$ and $\bar{V}'$ have no attributes in common. It can be seen that when evaluating the LARA expression on a LARA database D, we obtain precisely the tuples $(\bar{k},\bar{v}) \in e^D_{\phi_1}$ such that there is a tuple of the form $(\bar{k},\bar{w}) \in e^D_{\phi_2}$ for which $\bar{v} \neq \bar{w}$. This means that e^D_α is precisely the set (P1), which by definition is included in e^D_ϕ. It is easy to see that, for our example above, the output of the expression

$$e_\alpha[K,V] := \bigvee^{K,V}_{\mathsf{Values}}\left[\mathsf{neq}_{V,V'}\big(e_{\phi_1} \bowtie \mathsf{rename}_{V\to V'}(e_{\phi_2})\big)\right],$$

is precisely $\{(k_2,1)\}$, i.e., the set of tuples $(k,v) \in e^D_{\phi_1}$ such that there is a tuple of the form $(k,w) \in e^D_{\phi_2}$ for which $v \neq w$.

Second, we consider the expression

$$e_1[\bar{K},\bar{V}] := (\bigvee^{\bar{K},\emptyset}_{\mathsf{Values}} e_{\phi_1}) \bowtie (e_{\phi_1} \bigvee_{\mathsf{func}_\oplus} e_{\phi_2}),$$

which takes the union of e_{ϕ_1} and e_{ϕ_2}, resolving conflicts with an aggregate operator $\mathsf{func}_\oplus$ that signals for which tuples of keys it requires to restore "key-functionality" after performing the union, and then takes a join with the set of keys that appear in e_{ϕ_1}. The aggregate operator $\mathsf{func}_\oplus$ takes as input a multiset of values: if it contains more than one element, it returns the neutral value $0_\oplus$; otherwise it returns the only element in the multiset. For instance, $\mathsf{func}(\{\!\{a,a\}\!\}) = 0_\oplus$ and $\mathsf{func}(\{\!\{a\}\!\}) = a$. Notice that e^D_1, for D a LARA database, contains the tuples $(\bar{k},\bar{v}) \in e^D_{\phi_1}$ such that there is no tuple of the form $(\bar{k},\bar{v}') \in e^D_{\phi_2}$, plus the tuples of the form $(\bar{k},0_\oplus,\ldots,0_\oplus)$ such that there are tuples of the form $(\bar{k},\bar{v}_1) \in e^D_{\phi_1}$ and $(\bar{k},\bar{v}_2) \in e^D_{\phi_2}$. In other words, by evaluating e_1 on D we have marked with $(0_\oplus,\ldots,0_\oplus)$ those tuples $\bar{k}$ of keys for which we need to restore "key-functionality" after performing the union. Then we take the expression

$$e_\beta := \mathsf{neq}_{\bar{V}=(0_\oplus,\ldots,0_\oplus)}(e_1),$$

which retrieves all tuples of the form $(\bar{k},\bar{v}) \in e^D_{\phi_1}$ such that (a) $\bar{v} \neq (0_\oplus,\ldots,0_\oplus)$, and (b) there is no tuple of the form $(\bar{k},\bar{v}') \in e^D_{\phi_2}$. This means that e^D_β is precisely the set (P2), which by definition is included in e^D_ϕ.

For our example above, the output of the expression $(e_{\phi_1} \bigvee_{\mathsf{func}_\oplus} e_{\phi_2})$ is

$$\{(k_1,0),(k_2,0),(k_3,2),(k_4,0),(k_5,7)\},$$

and that of e_1 is $\{(k_1,0),(k_2,0),(k_3,2),(k_4,0)\}$. Hence, the output of e_β is $\{(k_3,2)\}$, i.e., the set of tuples $(k,v) \in e^D_{\phi_1}$ such that $v \neq 0$ and there is no tuple of the form $(k,w) \in e^D_{\phi_2}$.

We now perform our last step. As mentioned, the evaluation of e_ϕ on D must contain both e^D_α and e^D_β. There is, however, a set of tuples that are still missing from e^D_ϕ; namely, those in (P3), i.e., those of the form $(\bar{k},0_\oplus,\ldots,0_\oplus) \in e^D_{\phi_1}$ for which there is no tuple of the form $(\bar{k},\bar{v}) \in e^D_{\phi_2}$. Let us define an expression

$$e_2[\bar{K},\bar{V}] := (\bigvee^{\bar{K},\emptyset}_{\mathsf{Values}} e_{\phi_1}) \bowtie (e_{\phi_1} \bigvee_{\mathsf{func}'_\oplus} e_{\phi_2}),$$

where $\mathsf{func}'_\oplus$ is an aggregate operator that takes as input a multiset of values: if it contains more than one element, it returns some value $1_\oplus$ with $1_\oplus \neq 0_\oplus$; otherwise it returns the neutral value $0_\oplus$. For instance, $\mathsf{func}(\{\!\{a,a\}\!\}) = 1_\oplus$ and $\mathsf{func}(\{\!\{a\}\!\}) = 0_\oplus$. Notice that the evaluation of e_2 on D, for D a LARA database, contains the tuples $(\bar{k},\bar{v}) \in e^D_{\phi_1}$ such that there is no tuple of the form $(\bar{k},\bar{v}') \in e^D_{\phi_2}$, plus the tuples of the form $(\bar{k},1_\oplus,\ldots,1_\oplus)$ such that there are tuples of the form $(\bar{k},\bar{v}) \in e^D_{\phi_1}$ and $(\bar{k},\bar{v}') \in e^D_{\phi_2}$. Then we take the expression

$$e_\gamma := \mathsf{eq}_{\bar{V}=(0_\oplus,\ldots,0_\oplus)}(e_1),$$

which retrieves exactly what we wanted: all tuples of the form $(\bar{k},0_\oplus,\ldots,0_\oplus) \in e^D_{\phi_1}$ such that there is no tuple of the form $(\bar{k},\bar{v}) \in e^D_{\phi_2}$.

For our example above, the output of the expression $(e_{\phi_1} \bigvee_{\mathsf{func}'_\oplus} e_{\phi_2})$ is

$$\{(k_1,1),(k_2,1),(k_3,2),(k_4,0),(k_5,7)\},$$

and that of e_2 is $\{(k_1,1),(k_2,1),(k_3,2),(k_4,0)\}$. Hence, the output of e_γ is $\{(k_4,0)\}$, i.e., the set of tuples $(k,v) \in e^D_{\phi_1}$ such that $v = 0$ and there is no tuple of the form $(k,w) \in e^D_{\phi_2}$.

Summing up, we can define $e_\phi := (e_\alpha \bowtie e_\beta) \bowtie e_\gamma$. There is no need to specify an aggregation operation in this case for $\bowtie$, since by definition the union of the three tables contains no conflicts on keys.

Sum and projection. In this case ϕ is of the form $\mathsf{Group}_\oplus(\pi_{\bar{K},\bar{V}}\phi_1 + \pi_{\bar{K},\bar{V}}\phi_2)$, where ϕ_1, ϕ_2 are constrained expressions. Let us assume that $e_{\phi_1}[\bar{K}_1, \bar{V}_1]$ and $e_{\phi_2}[\bar{K}_2, \bar{V}_2]$ are the LARA expressions for ϕ_1 and ϕ_2, respectively, where $\bar{K} \subseteq \bar{K}_1 \cap \bar{K}_2$ and $\bar{V} \subseteq \bar{V}_1 \cap \bar{V}_2$. Then we can define a LARA expression for ϕ as follows:

$$e_\phi := \bowtie^{\bar{K},\bar{V}}_{\mathsf{Values}} \big(\mathsf{rename}_{\bar{K}_1\cap\bar{K}_2\setminus\bar{K}\to\bar{K}'}(e_{\phi_1}) \bowtie_\oplus e_{\phi_2}\big).$$

In fact, by definition the evaluation of the expression $\big(\mathsf{rename}_{\bar{K}_1\cap\bar{K}_2\setminus\bar{K}\to\bar{K}'}(e_{\phi_1}) \bowtie_\oplus e_{\phi_2}\big)$ over a LARA database D is

$$\mathsf{Solve}_\oplus\{\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in e^D_{\phi_1},\ \bar{k} = \bar{k}_1{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \mathrm{pad}^{\bar{V}_2}_\oplus(\bar{v}_1),\ \text{or } (\bar{k}_2, \bar{v}_2) \in e^D_{\phi_2},\ \bar{k} = \bar{k}_2{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \mathrm{pad}^{\bar{V}_1}_\oplus(\bar{v}_2)\}\}.$$

Hence, the evaluation of e_ϕ on D corresponds to the associative table

$$\mathsf{Solve}_\oplus\{\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in e^D_{\phi_1},\ \bar{k} = \bar{k}_1{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \bar{v}_1{\downarrow_{\bar{V}}},\ \text{or } (\bar{k}_2, \bar{v}_2) \in e^D_{\phi_2},\ \bar{k} = \bar{k}_2{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \bar{v}_2{\downarrow_{\bar{V}}}\}\}.$$

By inductive hypothesis, the latter is equivalent to

$$\mathsf{Solve}_\oplus\{\{(\bar{k}, \bar{v}) \mid (\bar{k}_1, \bar{v}_1) \in \phi^D_1,\ \bar{k} = \bar{k}_1{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \bar{v}_1{\downarrow_{\bar{V}}},\ \text{or } (\bar{k}_2, \bar{v}_2) \in \phi^D_2,\ \bar{k} = \bar{k}_2{\downarrow_{\bar{K}}},\ \text{and } \bar{v} = \bar{v}_2{\downarrow_{\bar{V}}}\}\},$$

which in turn is equivalent to

$$\mathsf{Solve}_\oplus\{\{(\bar{k}, \bar{v}) \mid (\bar{k}, \bar{v}) \in (\pi_{\bar{K},\bar{V}}\phi_1)^D \text{ or } (\bar{k}, \bar{v}) \in (\pi_{\bar{K},\bar{V}}\phi_2)^D\}\}.$$

By definition, the latter is precisely ϕ^D.

For our example above, the output of the expression $e_{\phi_1} \bowtie_\oplus e_{\phi_2}$, assuming $\oplus$ to be the standard sum operation on $\mathbb{N}$, is $\{(k_1, 0), (k_2, 3), (k_3, 2), (k_4, 0), (k_5, 7)\}$. In this case, this is the same as the evaluation of e_ϕ as neither the rename nor the aggregation operation performed afterwards have an effect on the result. This coincides with our desired result with respect to ϕ, i.e., this set is precisely the one obtained by applying $\mathsf{Group}_\oplus$ over the sum of $e^D_{\phi_1}$ and $e^D_{\phi_2}$. □

6. Expressiveness of LARA in terms of ML operators

In this section we initiate our study of how the class of extension functions allowed to be used in LARA affects the expressiveness of the language. As our main conceptual contribution, we show that a large and relevant class of extension functions can be expressed directly in a well-studied extension of $\mathrm{RA}_{\mathsf{Agg}}$ which is amenable for theoretical exploration. In particular, this language can express all the extension functions defined in Section 4.2 that are used in our expressive completeness result presented before. Moreover, by using well-known properties of this language, we can show that it cannot express some relevant ML operations such as matrix convolution and inverse.

6.1. Discussion on numerical operations over the domain

In the rest of the section we assume that $\mathsf{Values} = \mathbb{Q}$. Since extension functions in Ω can a priori be arbitrary, to understand what LARA can express we first need to specify which classes of functions are allowed in Ω. In rough terms, this is determined by the operations that one can perform when comparing keys and values, respectively, as explained next.

- Extensions of relational algebra with aggregate operators over a *numerical* sort $\mathcal{N}$ often permit to perform arbitrary numerical comparisons over $\mathcal{N}$ (in our case $\mathcal{N} = \mathsf{Values} = \mathbb{Q}$). It has been noted that this extends the expressive power of the language, but at the same time preserves some properties of the language that allow to carry out an analysis of its expressiveness based on well-established techniques (see, e.g., [19]).
- In some cases in which the expressive power of the language needs to be further extended, one can also define a linear order on the non-numerical sort (which in our case is the set Keys) and then perform suitable arithmetic comparisons in terms of such a linear order. A well-known application of this idea is in the area of descriptive complexity [15].

In this section we study the first possibility only. That is, we allow comparing elements of Values (i.e., $\mathbb{Q}$) in terms of arbitrary numerical operations. Elements of Keys, in turn, can only be compared with respect to equality. As we explain below, this class of extension functions is amenable for theoretical exploration – in particular, in terms of its expressive power – and at the same time is able to express many extension functions of practical interest (e.g., several of the functions used in examples in [13,14]).

6.2. A language for expressing extension functions

We design a simple language $\mathrm{EF}^{\mathsf{All}}_{=}$ for expressing extension functions based on the previous idea. Intuitively, the name of this language states that it can only compare keys with respect to equality $=$ but it can compare values in terms of arbitrary *(all)* numerical predicates. The expressions in the language are defined as follows, by distinguishing between key- and value-expressions and then allowing for the combination of both in a controlled fashion:

- The expressions $K_1 = K_2$ and $K_1 \neq K_2$, for K_1, K_2 different key-attributes, are the *key-expressions* of $\mathrm{EF}^{\mathsf{All}}_{=}$. These expressions are of sort (K_1, K_2).
- The *value-expressions* of $\mathrm{EF}^{\mathsf{All}}_{=}$ are constructed by using standard relational algebra operations over all possible expressions $P(V_1, \ldots, V_k)$, for $P \subseteq \mathbb{Q}^k$ a numerical relation of arity k and $V_1, \ldots, V_k$ pairwise different value-attributes. The expressions of the form $P(V_1, \ldots, V_k)$ are of sort $(V_1, \ldots, V_k)$.
- Key- and value-expressions of $\mathrm{EF}^{\mathsf{All}}_{=}$ form the set of *atomic expressions* of $\mathrm{EF}^{\mathsf{All}}_{=}$.
- If ϕ, ϕ' are expressions of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$ in $\mathrm{EF}^{\mathsf{All}}_{=}$, respectively, then $\phi \bowtie \phi'$ is an expression of sort $(\bar{K}_1 \cup \bar{K}_2, \bar{V}_1 \cup \bar{V}_2)$ in $\mathrm{EF}^{\mathsf{All}}_{=}$.
- If ϕ, ϕ' are expressions of sort $(\bar{K}, \bar{V})$ in $\mathrm{EF}^{\mathsf{All}}_{=}$, then $\phi \cup \phi'$ is an expression of sort $(\bar{K}, \bar{V})$ in $\mathrm{EF}^{\mathsf{All}}_{=}$.

We write $\phi(\bar{K}, \bar{V})$ to denote that the expression ϕ in $\mathrm{EF}^{\mathsf{All}}_{=}$ is of sort $(\bar{K}, \bar{V})$. Given a tuple t of the same sort as ϕ, we say that t *satisfies* ϕ if the following hold (omitting the rules for value-expressions formed by using relational algebra operations on top of the numerical predicates):

- ϕ is $K_1 = K_2$ and $t(K_1) = t(K_2)$, or ϕ is $K_1 \neq K_2$ and $t(K_1) \neq t(K_2)$.
- ϕ is $P(V_1, \ldots, V_k)$, for $P \subseteq \mathbb{Q}^k$ a numerical relation of arity k, and t belongs to the interpretation of P over Q^k.
- ϕ is $\phi_1 \bowtie \phi_2$, for ϕ_1 and ϕ_2 of sort $(\bar{K}_1, \bar{V}_1)$ and $(\bar{K}_2, \bar{V}_2)$, respectively, and $t = (\bar{k}_1 \cup \bar{k}_2, \bar{v}_1 \cup \bar{v}_2)$ for $(\bar{k}_1, \bar{v}_1)$ and $(\bar{k}_2, \bar{v}_2)$ tuples that satisfy ϕ_1 and ϕ_2, respectively.
- ϕ is $\phi_1 \cup \phi_2$, for ϕ_1 and ϕ_2 of sort $(\bar{K}, \bar{V})$, and t satisfies ϕ_1 or ϕ_2.

Definition 5 *(Definability of extension functions).* An extension function f of sort $(\bar{K}, \bar{V}) \mapsto (\bar{K}', \bar{V}')$ is *definable* in the language $\mathrm{EF}^{\mathsf{All}}_{=}$, if there is an expression $\phi_f(\bar{K}, \bar{K}', \bar{V}, \bar{V}')$ of $\mathrm{EF}^{\mathsf{All}}_{=}$ such that for every tuple $(\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V})$:

$$f(\bar{k}, \bar{v}) \;=\; \{(\bar{k}', \bar{v}') \mid (\bar{k}, \bar{k}', \bar{v}, \bar{v}') \text{ satisfies } \phi_f\}.$$

This gives rise to the definition of the following class of extension functions:

$$\Omega^{\mathsf{All}}_{=} \;=\; \{f \mid f \text{ is an extension function definable in } \mathrm{EF}^{\mathsf{All}}_{=}\}.$$

Recall that extension functions only produce finite associative tables by definition, and hence only some expressions in $\mathrm{EF}^{\mathsf{All}}_{=}$ define extension functions.

It is relatively easy to see that all the extension functions defined in Section 4.2, and used in the expressive completeness result, are in $\Omega^{\mathsf{All}}_{=}$. Next we provide more examples.

Example 3. We use $i + j = k$ and $(i + j)/2 = k$ as a shorthand notation for the ternary numerical predicates of addition and average, respectively. Consider first an extension function f that takes a tuple t of sort (K_1, K_2, V) and computes a tuple t' of sort (K_1', K_2', V') such that $t(K_1, K_2) = t'(K_1', K_2')$ and $t'(V') = 1 - t(V)$. Then f is definable in $\mathrm{EF}^{\mathsf{All}}_{=}$ as

$$\phi_f(K_1, K_2, K_1', K_2', V, V') \;:=\; (K_1 = K_1') \bowtie (K_2 = K_2') \bowtie (V + V' = 1).$$

This function can be used, e.g., to interchange 0s and 1s in a Boolean matrix.

Consider now an extension function g that takes a tuple t of sort (K, V_1, V_2) and computes a tuple t' of sort (K', V') such that $t(K) = t'(K')$ and $t'(V)$ is the average between $t(V_1)$ and $t(V_2)$. Then g is definable in $\mathrm{EF}^{\mathsf{All}}_{=}$ as

$$\phi_g(K, K', V_1, V_2, V') \;:=\; (K = K') \bowtie ((V_1 + V_2)/2 = V'). \quad \square$$

As an immediate corollary to Theorem 1 we obtain the following result, which formalizes the fact that for translating $\textsc{Lara}(\Omega^{\mathsf{All}}_{=})$ expressions into $\mathrm{RA}_{\mathsf{Agg}}$ it is not necessary to extend the expressive power of $\mathrm{RA}_{\mathsf{Agg}}$ with the relations in $\Psi_{\Omega^{\mathsf{All}}_{=}}$; in fact, it suffices in this case to grant access to all numerical predicates over $\mathbb{Q}$. Formally, let us denote by $\mathrm{RA}_{\mathsf{Agg}}(\mathsf{All})$ the extension of $\mathrm{RA}_{\mathsf{Agg}}$ with all expressions of the form $P(V_1, \ldots, V_k)$, for $P \subseteq \mathbb{Q}^k$ and $V_1, \ldots, V_k$ pairwise different value-attributes, with the expected semantics. Then one can prove the following result.

Corollary 1. *For every expression $e[\bar{K}, \bar{V}]$ of $\textsc{Lara}(\Omega^{\mathsf{All}}_{=})$ there is a constrained expression $\phi_e(\bar{K}, \bar{V})$ of $\mathrm{RA}^{c}_{\mathsf{Agg}}(\mathsf{All})$ with $e^D = \phi_e^D$, for each* $\textsc{Lara}$ *database D.*

6.3. Non-definability in $\text{LARA}(\Omega_{=}^{\text{All}})$

It is known that queries definable in $\text{RA}_{\text{Agg}}(\text{All})$ satisfy two important properties, namely, *genericity* and *locality*. These properties allow us to prove that neither convolution of matrices nor matrix inversion can be defined in the language. From Corollary 1 we obtain then that none of these queries is expressible in $\text{LARA}(\Omega_{=}^{\text{All}})$. We explain this next.

Convolution. Let A be an arbitrary matrix and N a square matrix. For simplicity we assume that N is of odd size $(2n+1) \times (2n+1)$. The convolution of A and N, written $A * N$, is a matrix of the same size as A with entries are defined as

$$(A * N)_{k\ell} = \sum_{s=1}^{2n+1} \sum_{t=1}^{2n+1} A_{k-n+s,\ell-n+t} \cdot N_{st}. \tag{5}$$

Notice that $k-n+s$ and $\ell-n+t$ could be invalid indices for matrix A. The standard way of dealing with this issue is *zero padding*. This simply assumes those entries outside A to be 0. In the context of the convolution operator, one usually calls N a *kernel*.

We represent the matrices A and N over the schema σ that consists of relation symbols $\{\text{Entry}_A[K_1, K_2, V], \text{Entry}_K[K_1, K_2, V]\}$. Assume that $\text{Keys} = \{\text{k}_1, \text{k}_2, \text{k}_3, \ldots\}$ and $\text{Values} = \mathbb{Q}$. If A is a matrix of values in $\mathbb{Q}$ of dimension $m \times p$, and N is a matrix of values in $\mathbb{Q}$ of dimensions $(2n+1) \times (2n+1)$ with $m, p, n \geq 1$, we represent the pair (A, N) as the LARA database $D_{A,N}$ over σ that contains all facts $\text{Entry}_A(\text{k}_i, \text{k}_j, A_{ij})$, for $i \in [m]$, $j \in [p]$, and all facts $\text{Entry}_K(\text{k}_i, \text{k}_j, N_{ij})$, for $i \in [2n+1]$, $j \in [2n+1]$. The query Convolution over schema σ takes as input a LARA database of the form $D_{A,N}$ and returns as output an associative table of sort $[K_1, K_2, V]$ that contains exactly the tuples $(\text{k}_i, \text{k}_j, (A * N)_{ij})$. We can then prove the following result.

Proposition 3. *The query* Convolution *is not expressible in* $\text{LARA}(\Omega_{=}^{\text{All}})$.

Proof. We first observe that when $\text{LARA}(\Omega_{=}^{\text{All}})$ expressions are interpreted as expressions over matrices, they are invariant under reordering of rows and columns of those matrices. More formally, we make use of *key-permutations* and *key-generic* queries. A *key-permutation* is an injective function $\pi : \text{Keys} \to \text{Keys}$. We extend a key-permutation π to be a function over $\text{Keys} \cup \text{Values}$ by letting π be the identity over Values. An expression $\phi(\bar{K}, \bar{V})$ is *key-generic* if for every LARA database D, key-permutation π, and tuple $t = (\bar{k}, \bar{v})$ of sort $(\bar{K}, \bar{V})$,

$$t \in \phi^D \quad \Leftrightarrow \quad \pi(t) \in \phi^{\pi(D)}$$

where $\pi(D)$ is the LARA database that is obtained by applying π on every fact of D. The following lemma expresses the self-evident property that expressions in $\text{RA}_{\text{Agg}}(\text{All})$ are key-generic.

Lemma 1. *Every expression* $\phi(\bar{K}, \bar{V})$ *of* $\text{RA}_{\text{Agg}}(\text{All})$ *is key-generic.*

With the aid of Lemma 1 we can now prove Proposition 3, as it is easy to show that Convolution is not key-generic (even when the kernel N is fixed). To obtain a contradiction assume that there exists an expression $\phi(K_1, K_2, V)$ in $\text{RA}_{\text{Agg}}(\text{All})$ such that for every LARA database $D_{A,N}$, as defined above, we have that $(\text{k}_i, \text{k}_j, v) \in \phi^{D_{A,N}}$ iff $(A * N)_{ij} = v$. Let A, N, and A' be the following matrices:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad N = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

$$A' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The LARA representations for these matrices are depicted in Fig. 6. Consider now the key-permutation π such that $\pi(\text{k}_2) = \text{k}_3$, $\pi(\text{k}_3) = \text{k}_2$, and π is the identity for every other value in Keys. Clearly then, $\pi(D_{A,N}) = D_{A',N}$. Now, the convolutions $(A * N)$ and $(A' * N)$ are given by the matrices

$$(A * N) = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 \\ 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix} \qquad (A' * N) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 \end{bmatrix}$$

$\mathsf{Entry}_A =$

K_1	K_2	V
k$_1$	k$_1$	1
k$_1$	k$_2$	0
⋮	⋮	⋮
k$_2$	k$_1$	0
k$_2$	k$_2$	1
k$_2$	k$_3$	0
⋮	⋮	⋮
k$_4$	k$_3$	0
k$_4$	k$_4$	1

$\mathsf{Entry}_N =$

K_1	K_2	V
k$_1$	k$_1$	1
k$_1$	k$_2$	1
⋮	⋮	⋮
k$_3$	k$_2$	1
k$_3$	k$_3$	1

$\mathsf{Entry}_{A'} =$

K_1	K_2	V
k$_1$	k$_1$	1
k$_1$	k$_2$	0
⋮	⋮	⋮
k$_3$	k$_2$	0
k$_3$	k$_3$	1
k$_3$	k$_4$	0
⋮	⋮	⋮
k$_4$	k$_3$	0
k$_4$	k$_4$	1

Fig. 6. LARA representations for matrices A and K in the proof of Proposition 3.

We know that $(\mathsf{k}_1, \mathsf{k}_1, 2) \in \phi^{D_{A,N}}$ (since $(A * N)_{11} = 2$). Then, since ϕ is generic, we have that $\pi(\mathsf{k}_1, \mathsf{k}_1, 2) \in \phi^{\pi(D_{A,N})}$. Thus, since $\pi(D_{A,N}) = D_{A',N}$, $\pi(\mathsf{k}_1) = \mathsf{k}_1$, and π is the identity over Values, we obtain that $(\mathsf{k}_1, \mathsf{k}_1, 2) \in \phi^{D_{A',N}}$, which is a contradiction since $(A' * N)_{11} = 1 \neq 2$. This proves that Convolution is not expressible in $\mathsf{RA}_{\mathsf{Agg}}(\mathsf{All})$. Hence from Corollary 1 we obtain that $\text{LARA}(\Omega_{\text{Agg}})$ cannot express Convolution. □

Matrix inverse. It has been shown by Brijder et al. [2] that matrix inversion is not expressible in MATLANG by applying techniques based on locality. The basic idea is that MATLANG is subsumed by $\mathsf{RA}_{\mathsf{Agg}}(\emptyset) = \mathsf{RA}_{\mathsf{Agg}}$, and the latter language can only define *local* properties. Intuitively, this means that expressions in $\mathsf{RA}_{\mathsf{Agg}}$ can only distinguish up to a *fixed-radius* neighborhood from its free variables (see, e.g., [19] for a formal definition). On the other hand, as shown in [2], if matrix inversion were expressible in MATLANG there would also be a $\mathsf{RA}_{\mathsf{Agg}}$ expression that defines the transitive closure of a binary relation (represented by its adjacency Boolean matrix). This is a contradiction as transitive closure is the prime example of a non-local property. Exactly the same techniques can be used to show that matrix inversion is not expressible in $\text{LARA}(\Omega_{=}^{\mathsf{All}})$. For this, we use first the fact that $\text{LARA}(\Omega_{=}^{\mathsf{All}})$ can be embedded in $\mathsf{RA}_{\mathsf{Agg}}(\mathsf{All})$ by Corollary 1, and then that $\mathsf{RA}_{\mathsf{Agg}}(\mathsf{All})$ is also local (cf., [11,19]).

As before, we represent Boolean matrices as databases over schema $\sigma = \{\mathsf{Entry}[K_1, K_2, V]\}$. Assume that $\mathsf{Keys} = \mathbb{N}$ and $\mathsf{Values} = \mathbb{Q}$. The Boolean matrix M of dimension $n \times m$, for $n, m \geq 1$, is represented as the LARA database D_M over σ that contains all facts $\mathsf{Entry}(i, j, b_{ij})$, for $i \in [n]$, $j \in [m]$, and $b_{ij} \in \{0, 1\}$, such that $M_{ij} = b_{ij}$. Consider the query Inv over schema σ that takes as input a LARA database of the form D_M and returns as output the LARA database $D_{M^{-1}}$, for M^{-1} the inverse of M. We can then obtain the following result by using the aforementioned argument.

Proposition 4. *The query* Inv *is not expressible in* $\text{LARA}(\Omega_{=}^{\mathsf{All}})$.

7. Adding built-in predicates over keys

7.1. Expressing the convolution query in LARA

In Section 6 we saw that there are important linear algebra operations, such as matrix inverse and convolution, that $\text{LARA}(\Omega_{=}^{\mathsf{All}})$ cannot express. The following result shows, in turn, that a clean extension of $\text{LARA}(\Omega_{=}^{\mathsf{All}})$ can express matrix convolution. This extension corresponds to the language $\text{LARA}(\Omega_{<}^{\mathsf{All}})$, i.e., the extension of $\text{LARA}(\Omega_{=}^{\mathsf{All}})$ in which we assume the existence of a strict linear-order $<$ on Keys and extension functions are definable in the logic $\mathsf{EF}_{<}^{\mathsf{All}}$ that extends $\mathsf{EF}_{=}^{\mathsf{All}}$ by allowing atomic formulas of the form $K_1 < K_2$ and $\neg(K_1 < K_2)$, for K_1, K_2 key-attributes. Even more, the only numerical predicates from All we need are $+$ and $\times$, as long as we have access to the extension functions for copying attributes and projection over value-attributes defined in Section 5 and used in our expressive completeness results. We denote the resulting logic as $\text{LARA}(\Omega_{<}^{\{+,\times\}})$.

Proposition 5. *The query* CONVOLUTION *is expressible in* $\text{LARA}(\Omega_{<}^{\{+,\times\}})$.

Proof of Proposition 5. We organize the proof in three parts. We first show how expressions in $\text{LARA}(\Omega_{<}^{\{+,\times\}})$ can express arithmetical operations over key-attributes, and then apply this idea to define the convolution query.

Expressing numerical operations over key- and value-attributes. Assume for simplicity that $\mathsf{Keys} = \mathbb{N}$ and $\mathsf{Values} = \mathbb{Q}$. Consider first the extension function $f : (K, K', \emptyset) \mapsto (\emptyset, V)$ defined as the following $\mathsf{EF}_{<}^{\{+,\times\}}$ expression:

$$\phi_f(K, K', V) \;:=\; \big(\neg(K < K') \cup \mathsf{Zero}(V)\big) \;\bowtie\; \big((K < K') \cup \mathsf{One}(V)\big),$$

where Zero and One are the numerical predicates $\{0\}$ and $\{1\}$, respectively. Notice that we slightly abuse terminology here as we allow unions of expressions with different sets of attributes, something that it is not permitted in $\mathsf{EF}_{<}^{\{+,\times\}}$. It is easy

to see, however, that they can be mimicked by adding suitable joins on the expressions. Now consider a relation Entry_A of sort $[K, K', V]$ that represents a square matrix of dimension $n \times n$. By our definition of f we have that $\mathsf{Map}_f\,\mathsf{Entry}_A$ has sort $[K, K', V]$ and its evaluation consists of all triples $(x, y, i) \in [n]^3$ such that $i = 1$ if $x \geq y$, and $i = 0$ otherwise.

Consider now the expression

$$\mathsf{Ind}_{K,V} \;:=\; \textstyle\sum_{+}^{K} \mathsf{Map}_f\,\mathsf{Entry}_A$$

that aggregates $\mathsf{Map}_f\,\mathsf{Entry}_A$ by summing over V while grouping over K. The evaluation of $\mathsf{Ind}_{K,V}$ contains all pairs $(x, i) \in [n] \times [n]$ such that i is the natural number that represents the position of x in the linear order over Keys. Hence, we have that $\mathsf{Ind}_{K,V}$ contains all pairs $(x, i) \in [n] \times [n]$ such that x is a key, i is a value, and $x = i$. This simple fact allows us to express filters and extension functions using arbitrary properties definable as an expression with predicates over key- and value-attributes together, using symbols $<$, $+$, and $\times$, and without actually mixing sorts.

For example, consider an associative table R of sort $[K, V_1]$ with $[n]$ as set of keys, and assume that we want to construct a new table R' of sort $[K, V_2]$ such that, for every tuple $(x, v) \in R$, relation R' contains the tuple (x, j) with $j = 2x + v$. We make use of the extension function $g : (K, V, V_1) \mapsto (\emptyset, V_2)$ defined by the expression $\phi_g(K, V, V_1, V_2) := (V_2 = 2V_1 + V)$. Notice that ϕ_g only mentions attributes of the second sort. We can construct R' as

$$R' := \mathsf{Map}_g(\mathsf{Ind}_{K,V} \bowtie R)$$

To see that this works, notice first that $\mathsf{Ind}_{K,V}$ and R have the same set of keys; namely, the set $[n]$. Moreover, given that $\mathsf{Ind}_{K,V}$ and R share no value-attribute, we do not need to specify any aggregate operator on the join of the expression $\mathsf{Ind}_{K,V} \bowtie R$. Notice, then, that the result of $\mathsf{Ind}_{K,V} \bowtie R$ is a table of sort $[K, V, V_1]$ that contains all tuples (x, i, v) such that $(x, i) \in \mathsf{Ind}_{K,V}$ and $(x, v) \in R$, or equivalently, all tuples (x, i, v) such that $(x, v) \in R$ and $x = i$. Then with $\mathsf{Map}_g(\mathsf{Ind}_{K,V} \bowtie_+ R)$ we generate all tuples (x, j) such that

$$(x, v) \in R, \quad x = i, \quad \text{and} \quad (j = 2i + v),$$

or equivalently, all tuples (x, j) such that $(x, v) \in R$, and $(j = 2x + v)$, which is what we wanted to obtain.

Hence, from now on if $\phi(\bar{K}, \bar{V}, \bar{V}')$ is an expression in EF over key and value-attributes together, using predicates $<$, $+$, and $\times$, we allow to write $\mathsf{Map}_\phi e$ to define the set of pairs $(\bar{k}, \bar{v}')$ such that $(\bar{k}, \bar{v})$ belongs to the evaluation of e_1, for some tuple $\bar{v}$ of value-attributes, and $(\bar{k}, \bar{v}, \bar{v}')$ satisfies ϕ.

In the construction we also make use of filtering expressions defined as follows. Given an expression $e_1[\bar{K}, \bar{V}]$ and an EF expression $\varphi(\bar{K}, \bar{V})$ over key and value-attributes using predicates $<$, $+$, and $\times$, then *filtering* e_1 with φ is a new expression $e_2 = \mathsf{Filter}_\varphi(e_1)$ that has sort $[\bar{K}, \bar{V}]$ and such that for every database D it holds that $(\bar{k}, \bar{v})$ is in e_2^D if and only if $(\bar{k}, \bar{v}) \in e_1^D$ and $(\bar{k}, \bar{v})$ satisfies φ. It is easy to see that the filter operator is expressible in $\textsc{Lara}(\Omega_<^{\{+,\times\}})$.

Expressing the convolution. We now have all the ingredients to express the convolution. We first write the convolution definition in a more suitable way so we can easily express the required sums. Let N be a kernel of dimensions $\mathsf{m} \times \mathsf{m}$ with m an odd number. First define mid as $\frac{\mathsf{m}-1}{2}$. Consider now a matrix A of dimension $\mathsf{n}_1 \times \mathsf{n}_2$. Now, for every $(\mathsf{i}, \mathsf{j}) \in [\mathsf{n}_1] \times [\mathsf{n}_2]$, one can write the following expression for $(A * N)_{\mathsf{ij}}$:

$$\begin{aligned}(A * N)_{\mathsf{ij}} = \sum \{\!\{\, A_{\mathsf{st}} \cdot N_{\mathsf{kl}} \mid\ & \mathsf{s} \in [\mathsf{n}_1], \mathsf{t} \in [\mathsf{n}_2], \mathsf{k}, \mathsf{l} \in [\mathsf{m}] \\ & \text{and } \mathsf{i} - \mathsf{mid} \leq \mathsf{s} \leq \mathsf{i} + \mathsf{mid} \\ & \text{and } \mathsf{j} - \mathsf{mid} \leq \mathsf{t} \leq \mathsf{j} + \mathsf{mid} \\ & \text{and } \mathsf{k} = \mathsf{s} - \mathsf{i} + \mathsf{mid} + 1 \\ & \text{and } \mathsf{l} = \mathsf{t} - \mathsf{j} + \mathsf{mid} + 1 \,\}\!\}. \end{aligned} \tag{6}$$

Now, in order to implement the above definition in $\textsc{Lara}(\Omega_<^{\{+,\times\}})$ we use the extension function diag and the filtering expressions $\mathsf{neighbors}$ and kernel, as defined below, where we assume that all attributes used in formulas correspond to key-attributes. We use a loose syntax here, as otherwise the presentation would become very cumbersome, but we remark that in fact all these expressions can be expressed in the language EF over key and value-attributes together, using predicates $<$, $+$, and $\times$. The existential quantifier is a shorthand for expressing projection on all value attributes save for the one which is quantified:

$$\mathsf{diag}(k, \ell, m) \;:=\; (k = \ell) \to m = 1 \;\bowtie\; \neg(k = \ell) \to m = 0$$

$$\begin{aligned}&\mathsf{neighbors}(i, j, s, t, m) \;:=\; \exists \mathrm{mid}\big(2 \times \mathrm{mid} = m - 1 \bowtie \\ &i - \mathrm{mid} \leq s \bowtie s \leq i + \mathrm{mid} \bowtie\; j - \mathrm{mid} \leq t \bowtie t \leq j + \mathrm{mid}\big)\end{aligned}$$

$$\mathsf{kernel}(i, j, k, \ell, s, t, m) := \exists \mathrm{mid}\big(2 \times \mathrm{mid} = m - 1 \bowtie$$
$$k = s - i + \mathrm{mid} + 1 \quad \bowtie \quad \ell = t - j + \mathrm{mid} + 1\big)$$

We note that the first two expressions are essentially mimicking the inequalities in (6). The last expression is intuitively defining the diagonal.

Let $\mathsf{Entry}_A[(i, j), (v)]$ and $\mathsf{Entry}_N[(k, \ell), (u)]$ be the two associative tables that represent a matrix A and the convolution kernel N, respectively. Here i, j, k represent key-attributes, while u, v represent value-attributes. We first construct an expression that computes the dimension of the kernel:

$$M = \bigtriangledown_{+}^{\emptyset} \ \mathsf{Map}_{\mathsf{diag}}\mathsf{Entry}_N.$$

By the definition of diag we have that $M[\emptyset, (m)]$ has no key attributes, but has a single value attribute m that contains one tuple storing the dimension of K. Now we proceed to take the join of Entry_A with itself, Entry_N, and M. For that we need to make a copy of Entry_A in which attributes (i, j, v) are renamed as (s, t, w). This expression is thus defined as:

$$C = \mathsf{Entry}_A \bowtie \mathsf{Entry}_N \bowtie (\mathsf{rename}_{(i,j,v),(s,t,w)}(\mathsf{Entry}_A)) \bowtie M.$$

Notice that since the expressions involved in this join share neither key- nor value-attributes, the join is simply computing the cartesian product of the associative tables represented by such expressions. Notice that such a cartesian product thus produces an associative table of sort $C[(i, j, k, \ell, s, t), (v, u, w, m)]$.

To finalize, we compute the following filters over C:

$$F = \mathsf{Filter}_{\mathsf{kernel}}(\mathsf{Filter}_{\mathsf{neighbors}}(C)).$$

We note that F has sort $F[(i, j, k, \ell, s, t), (v, u, w, m)]$ (just like C). We also note that for every (i, j) the tuple $(\mathrm{i}, \mathrm{j}, \mathrm{k}, \mathrm{l}, \mathrm{s}, \mathrm{t})$ is a key in F if, and only, if it satisfies the conditions defining the multiset in Equation (6). Thus to compute what we need, it only remains to multiply and sum, which is done in the following expression:

$$R = \bigtriangledown_{+}^{ij} \ (\mathsf{Map}_{v^\star = w \times u} F).$$

Thus R is of sort $[(i, j), (v^\star)]$ and is such that $(\mathrm{i}, \mathrm{j}, \mathrm{v})$ is in R if and only if $\mathrm{v} = (A * K)_{\mathrm{ij}}$. □

Hutchison et al. [13] showed that for every fixed kernel N, the query $(A * N)$ is expressible in Lara. However, the Lara expression they construct depends on the values of N, and hence their construction does not show that in general convolution is definable in Lara. Our construction is stronger, as we show that there exists a *fixed* Lara($\Omega_{<}^{\{+,\times\}}$) expression that takes A and N as input and produces $(A * N)$ as output.

7.2. Can Lara($\Omega_{<}^{\{+,\times\}}$) *express inverse?*

We believe that Lara($\Omega_{<}^{\{+,\times\}}$) cannot express Inv. However, this seems quite challenging to prove. First, the tool we used for showing that Inv is not expressible in Lara($\Omega_{=}^{\mathsf{All}}$), namely, locality, is no longer valid in this setting. In fact, queries expressible in Lara($\Omega_{<}^{\{+,\times\}}$) are not necessarily local.

Proposition 6. *The language* Lara($\Omega_{<}^{\{+,\times\}}$) *can express non-local queries.*

The reason why this result holds is as follows. By the discussion in the proof of Proposition 5, one can use arbitrary predicates $+$ and $\times$ over Keys to define extension functions. With this observation one can construct, given a relation $A[K]$ that contains all values in $[n]$, a Lara($\Omega_{<}^{\{+,\times\}}$) expression that defines a relation $\mathsf{BIT}[K, V]$ that contains all pairs (x, i) such that $x \in [n]$ and the i-th bit of the binary expansion of x is 1. With BIT one can mimic the construction of Proposition 8.22 in [19] to show that Lara($\Omega_{<}^{\{+,\times\}}$) can express a nonlocal query. This construction is carried out in first-order logic extended with numerical predicates $+$ and $\times$, and hence in RA extended with numerical predicates $+$ and $\times$. The expressions used in the proof can then be mimicked in Lara($\Omega_{<}^{\{+,\times\}}$), using ideas developed in our expressive completeness result.

The proposition implies that one would have to apply techniques more specifically tailored for the logic, such as *Ehrenfeucht-Fraïssé* games, to show that Inv is not expressible in Lara($\Omega_{<}^{\{+,\times\}}$). Unfortunately, it is often combinatorially difficult to apply such techniques in the presence of built-in predicates, e.g., a linear order, on the domain; cf., [5,23,9]. So far, we have not managed to succeed in this regard.

In turn, we can show that Inv is not expressible in a natural restriction of Lara under complexity-theoretic assumptions. To start with, Inv is complete for the complexity class Det, which contains all those problems that are logspace reducible to computing the *determinant* of a matrix. It is known that Logspace ⊆ Det, where Logspace is the class of functions computable in logarithmic space, and this inclusion is believed to be proper [3]. Hence, under the assumption that Logspace ≠ Det, no language that can be evaluated in Logspace in *data complexity* can express the query Inv. Recall that a language

$\mathcal{L}$ can be evaluated in LOGSPACE in data complexity, if for each *fixed* expression e in $\mathcal{L}$ the evaluation e^D of e on an input LARA database D can be computed in LOGSPACE.

There is an extension of RA, known as RA *with arithmetic* [10], that has the property that can be evaluated in LOGSPACE in *data complexity*. In addition, it is able to express most aggregate operators used in practice, e.g., SUM, MIN, MAX, AVG, and COUNT. It is easy to see that RA with arithmetic subsumes the fragment of LARA, without extension functions, in which only these aggregate operators are used. If we now extend this fragment of LARA with any set of extension functions that can be computed in LOGSPACE, the resulting language can still be evaluated in LOGSPACE in *data complexity*. From our previous observation, under the assumption that LOGSPACE $\neq$ DET this fragment of LARA cannot express the matrix inversion query INV.

8. Final remarks and future work

We believe that the work on query languages for analytics systems that integrate relational and statistical functionalities provides interesting perspectives for database theory. In this paper we focused on the LARA language, which has been designed to become the core algebraic language for such systems, and carried out a systematic study of its expressive power in terms of logics and concepts traditionally studied in the database theory literature.

As we have observed, expressing interesting ML operators in LARA requires the addition of complex features, such as arithmetic predicates on the numerical sort and built-in predicates on the domain. The presence of such features complicates the study of the expressive power of the languages, as some known techniques no longer hold, e.g., genericity and locality, while others become combinatorially difficult to apply, e.g., Ehrenfeucht-Fraïssé games. In addition, the presence of a built-in linear order might turn the logic capable of characterizing some parallel complexity classes, and thus inexpressibility results could be as hard to prove as some longstanding conjectures in complexity theory.

A possible way to overcome these problems might be not looking at languages in its full generality, but only at extensions of the tame fragment LARA($\Omega^{\mathsf{All}}_{=}$) with some of the most sophisticated operators. For instance, what if we extend LARA($\Omega^{\mathsf{All}}_{=}$) directly with an operator that computes Convolution? Is it possible to prove that the resulting language (LARA($\Omega^{\mathsf{All}}_{=}$) + Convolution) cannot express matrix inverse Inv? Somewhat a similar approach has been followed in the study of MATLANG; e.g., [2] studies the language (MATLANG + INV), which extends MATLANG with the matrix inverse operator.

Another interesting line of work corresponds to identifying which kind of operations need to be added to LARA in order to be able to express in a natural way recursive operations such as matrix inverse. One would like to do this in a general yet minimalistic way, as adding too much recursive expressive power to the language might render it impractical. It would be important to start then by identifying the most important recursive operations one needs to perform on associative tables, and then abstract from them the minimal primitives that the language needs to possess for expressing such operations. A good starting point for this might be the recently proposed extension of MATLANG with recursive features [7].

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

Barceló and Pérez are funded by FONDECYT grant 1200967. All authors have been funded by ANID - Millennium Science Initiative Program - Code ICN17002. Barceló has also been funded by National Center for Artificial Intelligence CENIA FB210017, Basal ANID. We are grateful to the reviewers of a preliminary version of this paper who provided us with relevant and constructive criticisms.

References

[1] Pablo Barceló, Nelson Higuera, Jorge Pérez, Bernardo Subercaseaux, On the expressiveness of LARA: a unified language for linear and relational algebra, in: ICDT, 2020, 6.
[2] Robert Brijder, Floris Geerts, Jan Van den Bussche, Timmy Weerwag, On the expressive power of query languages for matrices, ACM Trans. Database Syst. 44 (4) (2019) 15.
[3] Stephen A. Cook, A taxonomy of problems with fast parallel algorithms, Inf. Control 64 (1–3) (1985) 2–21.
[4] Serge Abiteboul, et al., Research directions for principles of data management (Dagstuhl perspectives workshop 16151), Dagstuhl Manifestos 7 (1) (2018) 1–29.
[5] Ronald Fagin, Larry J. Stockmeyer, Moshe Y. Vardi, On monadic NP vs. monadic co-np, Inf. Comput. 120 (1) (1995) 78–92.
[6] Floris Geerts, On the expressive power of linear algebra on graphs, in: ICDT, 2019, 7.
[7] Floris Geerts, Thomas Muñoz, Cristian Riveros, Domagoj Vrgoc, Expressive power of linear algebra query languages, CoRR, arXiv:2010.13717, 2020.
[8] Goetz Graefe, Richard L. Cole, Fast algorithms for universal quantification in large databases, ACM Trans. Database Syst. 20 (2) (1995) 187–236.
[9] Martin Grohe, Thomas Schwentick, Locality of order-invariant first-order formulas, in: MFCS, 1998, pp. 437–445.
[10] Stéphane Grumbach, Leonid Libkin, Tova Milo, Limsoon Wong, Query languages for bags: expressive power and complexity, SIGACT News 27 (2) (1996) 30–44.
[11] Lauri Hella, Leonid Libkin, Juha Nurmonen, Limsoon Wong, Logics with aggregate operators, J. ACM 48 (4) (2001) 880–907.

[12] Stephan Hoyer, Joe Hamman, xarray developers, xarray development roadmap, Technical report, 2018, available at http://xarray.pydata.org/en/stable/roadmap.html.
[13] Dylan Hutchison, Bill Howe, Dan Suciu, Lara: a key-value algebra underlying arrays and relations, CoRR, arXiv:1604.03607, 2016.
[14] Dylan Hutchison, Bill Howe, Dan Suciu, Laradb: a minimalist kernel for linear and relational algebra computation, in: BeyondMR@SIGMOD 2017, 2017, 2.
[15] Neil Immerman, Descriptive Complexity, Graduate Texts in Computer Science, Springer, 1999.
[16] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandala, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, Carlo Curino, Extending relational query processing with ML inference, CoRR, arXiv:1911.00231, 2019.
[17] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, Volker Markl, Bridging the gap: towards optimization across linear and relational algebra, in: BeyondMR@SIGMOD 2016, 2016, p. 1.
[18] Leonid Libkin, Expressive power of SQL, Theor. Comput. Sci. 296 (3) (2003) 379–404.
[19] Leonid Libkin, Elements of Finite Model Theory, Texts in Theoretical Computer Science. An EATCS Series, Springer, 2004.
[20] Dan Olteanu, The relational data borg is learning, Proc. VLDB Endow. 13 (12) (2020) 3502–3515.
[21] Alexander M. Rush, Tensor considered harmful, Technical report, Harvard NLP Blog, 2019, available at http://nlp.seas.harvard.edu/NamedTensor, retrieved on March 2019.
[22] Alexander M. Rush, Tensor considered harmful pt. 2, Technical report, Harvard NLP Blog, 2019, available at http://nlp.seas.harvard.edu/NamedTensor2, retrieved on March 2019.
[23] Thomas Schwentick, On winning Ehrenfeucht games and monadic NP, Ann. Pure Appl. Log. 79 (1) (1996) 61–92.