Expressiveness of Matrix and Tensor Query Languages in terms of ML Operators

Pablo Barceló Nelson Higuera Jorge Pérez Bernardo Subercaseaux
Department of Computer Science, University of Chile & IMFD Chile

ABSTRACT

Tensors are one of the most widely used data structures in modern Machine Learning applications. Although they provide a flexible way of storing and accessing data, they often expose too many low-level details that may result in error prone code that is difficult to maintain and extend. Abstracting low-level functionalities into high-level operators in the form of a query language is a task in which the Data Management community has extensive experience. It is thus important to understand how such an experience can be applied in the design of useful languages for tensor manipulation.

In this short paper we study a matrix and a tensor query language that have been recently proposed in the database literature. We show, by using examples, how these proposals are in line with the practical interest in rethinking tensor abstractions. On the technical side, we compare the two languages in terms of operators that naturally arise in Machine Learning pipelines, such as convolution, matrix-inverse, and Einstein summation. We hope our results to provide a theoretical kick-off for the discussion on the design of core declarative query languages for tensors.

ACM Reference Format:
Pablo Barceló Nelson Higuera Jorge Pérez Bernardo Subercaseaux. . Expressiveness of Matrix and Tensor Query Languages in terms of ML Operators. In . ACM, New York, NY, USA, 5 pages.

1 INTRODUCTION

Matrices, and more generally tensors (multidimensional arrays of data), are ubiquitous in Machine Learning (ML) applications. Despite their wide adoption, there has been a recent interest in redesigning the way in which tensors are used in Deep Learning code [4, 18, 19]. In the entry *Tensor Considered Harmful* [18], Rush enumerates some pitfalls of the current way in which tensors are abstracted, stating that it *"forces bad habits such as exposing private dimensions, broadcasting based on absolute position, and keeping type information in documentation"*. From a data management point of view, this can be read as a call for a high-level language that manipulates tensors. In this paper we explore some recent proposals for matrix and tensor query languages, motivated by the need to understand their theoretical properties in terms of the functionalities required in ML applications.

Several papers have dealt with the implementation of tensor-based tasks using data management technologies [9–11, 16, 21], while others have studied how ML pipelines can be specified *in-database* [7, 8, 12]. Here, instead, we focus on what declarative database query languages can do directly over tensors in order to tackle their aforementioned pitfalls. We focus on two such languages, MATLANG [1] and LARA [6], as they were both defined with a precise syntax and formal semantics, and follow a minimalistic approach by introducing a small set of core operators. This makes them more amenable for theoretical exploration.

MATLANG is a language for matrix manipulation that resembles the syntax of standard linear algebra. LARA [6] consists of only three operators that provide a unified framework for linear and relational algebra. In what follows we introduce MATLANG and LARA by using an intuitive presentation based on examples. Then in Section 3 we formally study their expressiveness. While the expressiveness of MATLANG and LARA has been considered in isolation, here we focus on comparing their expressiveness in terms of some popular ML operators. For the sake of space proofs are presented in an appendix available at http://dcc.uchile.cl/~jperez/papers/deem19_sub.pdf.

2 MATLANG AND LARA

MATLANG. A schema for MATLANG expressions consists of a finite set of *matrix variables* and a mapping from any such variable A to a *type definition* that specifies the dimension of A. For example, if variable A has type (α, α), then A represents a square matrix. We use a type of the form $(\alpha, \mathbf{1})$ for variables representing column vectors, and analogously, $(\mathbf{1}, \alpha)$ for rows and $(\mathbf{1}, \mathbf{1})$ for scalars.

A MATLANG *expression* is a combination of matrix variables by means of linear algebra operators. As for matrix variables, every expression has a type that is induced from the types of its operands. Formally, the syntax of MATLANG is defined by induction as follows. If φ_1 and φ_2 are MATLANG expressions of types (α_1, β_1) and (α_2, β_2), respectively, then the following are also expressions:

- φ_1^T is an expression of type (β_1, α_1) (transposition).
- $1(\varphi_1)$ is an expression of type $(\alpha_1, \mathbf{1})$ (column of ones).
- $\mathsf{diag}(\varphi_1)$ is an expression of type (α_1, α_1) provided that $\beta_1 = \mathbf{1}$, i.e., φ_1 is a column vector (diagonalization).
- $\varphi_1 \cdot \varphi_2$ is an expression of type (α_1, β_2) provided that $\beta_1 = \alpha_2$ (matrix multiplication).
- $\mathsf{apply}[f](\varphi_1)$ where f is a function, is an expression of type (α_1, β_1) (pointwise function application)
- $\mathsf{apply}[\circ](\varphi_1, \varphi_2)$ where $\circ$ is a binary operation, is an expression of type (α_1, β_1) provided that $(\alpha_1, \beta_1) = (\alpha_2, \beta_2)$ (pointwise operation of matrices).

Every MATLANG expression is evaluated over a *matrix database instance*. An instance $\mathcal{I}$ of schema $\mathcal{S}$ assigns to every variable $A \in \mathcal{S}$ a matrix $\mathcal{I}(A)$ that complies with the type of A. Given an expression φ and an instance $\mathcal{I}$ we denote by $\varphi(\mathcal{I})$ the evaluation of φ over $\mathcal{I}$. The evaluation $\varphi(\mathcal{I})$ can easily be defined by using the linear-algebra definition of every operator above [1].

A *query* Q over a schema $\mathcal{S}$ is a function that assigns to every instance $\mathcal{I}$ of $\mathcal{S}$ an *output matrix* $Q(\mathcal{I})$. A query Q is expressible in MATLANG if there exists a MATLANG expression φ such that for every instance $\mathcal{I}$ of $\mathcal{S}$ it holds that $\varphi(\mathcal{I}) = Q(\mathcal{I})$. Brijder et al. [1] studied the expressive power of MATLANG over Boolean matrices, showing that in spite of the simplicity of the language it can express non-trivial properties including: (i) all queries that can be expressed in first-order logic with only three variables over binary matrices, and (ii) aggregations such as computing the mimimum value of a vector or the sum of all elements of a matrix.

Example 2.1 (From [1]). Let us define an operator $\circ_\leq$ such that $a \circ_\leq b = 1$ if $a \leq b$ and it is 0 otherwise. Analogously define operator $\circ_=$. The following expressions take a column vector v and compute as output a vector that has a 1 exactly on those positions of v that store its minimum value.

$$\varphi_1 = \mathsf{v} \cdot 1(\mathsf{v})^T \quad (1)$$

$$\varphi_2 = \mathsf{apply}[\circ_\leq](\varphi_1, \varphi_1^T) \cdot 1(\mathsf{v}) \quad (2)$$

$$\varphi_3 = 1(\mathsf{v}) \cdot 1(\mathsf{v})^T \cdot 1(\mathsf{v}) \quad (3)$$

$$\varphi_4 = \mathsf{apply}[\circ_=](\varphi_2, \varphi_3) \quad (4)$$

Expression (1) replicates v in the columns of a square matrix. Then expression (2) is used to compare every value in v with all the other values. It does so by first producing a binary square matrix ($\mathsf{apply}[\circ_\leq](\varphi_1, \varphi_1^T)$), which is then multiplied with $1(\mathsf{v})$. This essentially performs a sum over the rows. Thus φ_2 represents a column vector storing, for each of the values of v, how many other values are grater than or equal to it. In expression (3) we produce a column vector storing in every component the length of v, and finally expression (4) performs a point-wise comparison. Thus, the vector produced as output of expression (4) has a 1 exactly in those positions of v that store its minimum value.

LARA. This is a language defined in terms of an algebra over *associative tables*. An associative table is a relation A with two disjoint sets of attributes: *key* and *value* attributes. We use $A[K, V]$ to denote an associative table with $K = \{K_1, \ldots, K_n\}$ as its key attributes and $V = \{V_1, \ldots, V_m\}$ as its value attributes. An associative table contains *named* tuples of the form $\{K_1 : k_1, \ldots, K_n : k_n, V_1 : v_1, \ldots, V_m : v_m\}$. To simplify the exposition, we assume an arbitrary order among key and value attributes, and thus the content of an associative table can be seen as a standard set of tuples of the form $((k_1, \ldots, k_n), (v_1, \ldots, v_m))$. Associative tables satisfy the following *key constraint*: if $(\bar{k}, \bar{v}_1) \in A$ and $(\bar{k}, \bar{v}_2) \in A$, then $\bar{v}_1 = \bar{v}_2$. Thus, for an associative table $A[K, V]$ and a tuple $(\bar{k}, \bar{v}) \in A$ we can safely denote $\bar{v} = A(\bar{k})$.

An associative table is a generalization of a tensor (and thus, also a generalization of a matrix). For example, a tensor T of rank 3, can be represented as an associative table with a single value attribute $\mathsf{A}[(I, J, K), (val)]$ such that $\mathsf{A}(i, j, k) = \mathsf{T}_{ijk}$. Notice that, as opposed to the typical tensor data structure used in libraries such as numpy [15] or (py)torch [2, 17], associative tables have named attributes and LARA takes advantage of those attribute names to define operators. As we will see, associative tables are more similar to *named tensors* [4, 18]. From now on we focus on associative tables with a single value attribute and call them named tensors for simplicity.

LARA is defined in terms of three main operators: *union*, *join*, and *extend* [6]. Here we introduce a modified (less general) version, but we note that our results in the next section hold in general. We refer the reader to [5, 6] for the complete set of operators. We consider the following operators:

- *Join* ($\bowtie_\otimes$): Given named tensors A and B with key attributes K_1 and K_2, respectively, and a binary operator $\otimes$, then $\mathsf{T} = (\mathsf{A} \bowtie_\otimes \mathsf{B})$ is a new tensor with key attributes $K = K_1 \cup K_2$, and such that $\mathsf{T}(\bar{k}_1 \cup \bar{k}_2) = \mathsf{A}(\bar{k}_1) \otimes \mathsf{B}(\bar{k}_2)$. Notice that $\bar{k}_1 \cup \bar{k}_2$ is a valid tuple only when $\bar{k}_1$ and $\bar{k}_2$ give the same values to their key attributes in common. Thus operator $\bowtie_\otimes$ resembles the natural join operator in relational algebra.
- *Aggregation* ($\maltese^L_{f(\cdot)}$): Given named tensor A with key attributes K, function f over sets of values, and set $L \subseteq K$, then $\mathsf{T} = (\maltese^L_{f(\cdot)}\, \mathsf{A})$ is a new named tensor with key attributes L and $\mathsf{T}(\bar{\ell}) = f(\{\mathsf{A}(\bar{k}) \mid \bar{\ell} \subseteq \bar{k}\})$.
- *Map* ($\mathsf{map}_{g(\cdot)}$): If A is a named tensor with key attributes K and g is a function, $\mathsf{T} = (\mathsf{map}_{g(\cdot)}\, \mathsf{A})$ is a tensor with key attributes K and $\mathsf{T}(\bar{k}) = g(\mathsf{A}(\bar{k}))$.

We also introduce the *reduction* operator as a useful syntactic variation of aggregation. Given a named tensor A with key attributes K, the reduction operation denoted by $(\bar{⧗}^{L}_{f(\cdot)}\ \mathsf{A})$ is defined as $(⧗^{K\setminus L}_{f(\cdot)}\ \mathsf{A})$. Thus, in the reduction we explicitly state the key attributes over which the aggregation is performed.

Example 2.2. Consider the named tensor

$$\mathsf{Seqs}[(\mathit{time}, \mathit{batch}, \mathit{features}), (\mathit{val})].$$

This is a typical structure obtained as the output of a recurrent neural network that processes input sequences. The structure stores a set of *features* obtained when processing input symbols from a sequence, one symbol at a *time*. For efficiency the network can simultaneously process a *batch* of examples and give a single tensor as output. Assume that, in order to make a prediction one wants to first obtain, for every example, the maximum value of every feature over the time steps, and then apply a *softmax* function. One can specify all this process in LARA as follows.

$$\mathsf{Max} \;=\; \bar{⧗}^{(\mathit{time})}_{\max(\cdot)}\ \mathsf{Seqs} \tag{5}$$

$$\mathsf{Exp} \;=\; \mathrm{map}_{\exp(\cdot)}\ \mathsf{Max} \tag{6}$$

$$\mathsf{SumExp} \;=\; \bar{⧗}^{(\mathit{features})}_{\mathrm{sum}(\cdot)}\ \mathsf{Exp} \tag{7}$$

$$\mathsf{Softmax} \;=\; \mathsf{Exp} \Join_{\div} \mathsf{SumExp} \tag{8}$$

Expression (5) performs an aggregation over the *time* attribute to obtain the new tensor $\mathsf{Max}[(\mathit{batch}, \mathit{features}), (\mathit{val})]$ such that $\mathsf{Max}(b, f) = \max_{u=\mathsf{Seqs}(t,b,f)} u$. That is, Max stores the maximum value over all time steps (for every feature of every example). Expression (6) applies a point-wise exponential function to obtain the tensor $\mathsf{Exp}[(\mathit{batch}, \mathit{features}), (\mathit{val})]$ such that $\mathsf{Exp}(b, f) = \exp(\mathsf{Max}(b, f))$. In expression (7) we apply another aggregation to compute the sum of the exponentials of all the (maximum) features. Thus we obtain the tensor $\mathsf{SumExp}[(\mathit{batch}), (\mathit{val})]$ such that

$$\mathsf{SumExp}(b) \;=\; \sum_{f} \mathsf{Exp}(b, f) \;=\; \sum_{f} \exp(\mathsf{Max}(b, f)).$$

Finally, expression (8) mimics a *broadcast by name* [18] by applying point-wise division over $\mathsf{Exp}[(\mathit{batch}, \mathit{features}), (\mathit{val})]$ and $\mathsf{SumExp}[(\mathit{batch}), (\mathit{val})]$. This defines the new tensor $\mathsf{Softmax}[(\mathit{batch}, \mathit{features}), (\mathit{val})]$ such that

$$\mathsf{Softmax}(b, f) \;=\; \frac{\mathsf{Exp}(b, f)}{\mathsf{SumExp}(b)} \;=\; \frac{\exp(\mathsf{Max}(b, f))}{\sum_{f'} \exp(\mathsf{Max}(b, f'))}.$$

Thus, we effectively compute the *softmax* of the vector of maximum features over time for every example in the batch.

The notions of queries and expressibility for LARA are similar to that of MATLANG.

3 WHAT CAN AND CANNOT BE EXPRESSED WITH THESE LANGUAGES

Next we study the expressive power of MATLANG and LARA in terms of three important operations for ML applications: convolution, matrix-inverse, and Einstein summation.

3.1 Convolution

Let A be an arbitrary matrix and K a square matrix. For simplicity we assume that K is of odd size $(2n + 1) \times (2n + 1)$. The convolution of A and K, denoted by $\mathsf{A} * \mathsf{K}$, is a matrix of the same size as A whose entries are defined as

$$(\mathsf{A} * \mathsf{K})_{ij} \;=\; \sum_{s=1}^{2n+1} \sum_{t=1}^{2n+1} \mathsf{A}_{i-n+s,\, j-n+t} \cdot \mathsf{K}_{st}.$$

Notice that $i - n + s$ and $j - n + t$ could be invalid indices for matrix A. The standard way of dealing with this issue, and the one that we use here, is *zero padding*. This simply assumes those entries outside A to be 0. In the context of the convolution operator, one usually calls K a *kernel*. Our first result shows that the convolution operator is not expressible in MATLANG, even for fixed kernels. The proof is based on a simple *genericity* property for the language that is not preserved by convolution. This property intuitively expresses that MATLANG expressions are invariant under reordering of rows and columns.

THEOREM 3.1. *Convolution is not expressible in* MATLANG. *Moreover, there exists a fixed matrix K such that the convolution with K is not expressible in* MATLANG.

On the other hand, the next result states that convolution is expressible in LARA. This is due to the fact that LARA is not generic in the form described above, as the language can express arithmetic comparisons over the indices of a matrix.

THEOREM 3.2. *Convolution is expressible in* LARA.

It is worth remarking that Hutchison et al. [5] showed that for every fixed kernel K, the query $(\mathsf{A} * \mathsf{K})$ is expressible in LARA. However, the LARA expression they construct depends on the values of K, and hence their construction does not show that in general convolution is expressible in LARA. Our result in Theorem 3.2 is stronger, as we prove that there exists a *fixed* LARA expression that takes A and K as input and produces $(\mathsf{A} * \mathsf{K})$ as output.

3.2 Inverse

Brijder et al. proved the following result.

PROPOSITION 3.3 (FROM [1]). *The language* MATLANG *cannot express matrix-inversion.*

The proof of this result is based on a *locality* argument for MATLANG. Intuitively, the locality of MATLANG expresses that every expression in the language "can only see until a fixed neighborhood of its free variables"' [13]. What Brijder et al. show is that if the inverse were expressible, then MATLANG would also be able to express the *reachability* query over undirected graphs. The latter, though, is not a local query, thus leading to a contradiction.

In view of this observation, Brijder et al. [1] added the matrix-inverse operator to MATLANG obtaining a language MATLANG + INV. This language continues being generic in the aforementioned sense, and thus we can prove that it does not express convolution.

Proposition 3.4. *The language* MATLANG + INV *cannot express convolution.*

For the case of LARA, we conjecture that matrix-inversion is also not expressible, but we still do not have a complete proof for this result. What we can actually prove is that if we restrict LARA to only be able to express arithmetical comparisons over the values of matrices, but not over its indices, then inverse is not expressible. The reason is that the resulting language, which we call tame-LARA, is local. Thus, the same argument above to show that inverse is not expressible in MATLANG applies.

Proposition 3.5. *Matrix-inversion is not expressible in* tame-LARA.

3.3 Einstein summation

Einstein summation notation is another popular operator on tensors that generalizes operations such as the inner product of vectors, matrix trace, and tensor product and contraction. It is based on implicitly specifying summation over indexes in a formula. For instance, let A and B be two matrices and consider the expression $\mathsf{A}_{ij}\mathsf{B}_{jk}$. When viewed as an Einstein summation, it is interpreted as a tensor T with two indexes (that is, a matrix), such that $\mathsf{T}_{ik} = \sum_j \mathsf{A}_{ij} \cdot \mathsf{B}_{jk}$, and thus, defines matrix product. In general, every repeated index in the expression implies a summation over such an index, while indexes that are not repeated are consider *free* and thus part of the output.

In modern tensor libraries [15, 17], Einstein summation is implemented as an even more general function `einsum` that gives a specific indexing to every input tensor, and specifies also the indexes expected in the output tensor. For simplicity, we focus on the case when `einsum` receives only two input tensors. It also receives a specification of the form $(\alpha, \beta \to \gamma)$ such that α and β are the indexing of the input tensors A and B, respectively, and $\gamma \subseteq \alpha \cup \beta$ is the indexing of the output. Every index in $\alpha \cup \beta$ that is not in γ is summed in the output. For instance, $\mathtt{einsum}((ij, jk \to ik), \mathsf{A}, \mathsf{B})$ is the multiplication of matrices A and B. As a more general example, consider tensors A and B of rank 3 and 4 respectively. Then the expression $\mathtt{einsum}((ijk, k\ell js \to \ell i), \mathsf{A}, \mathsf{B})$ produces a tensor T of rank 2 such that $\mathsf{T}_{\ell i} = \sum_j \sum_k \sum_s \mathsf{A}_{ijk} \cdot \mathsf{B}_{k\ell js}$. Notice that γ can be the empty set, in which case the output is a scalar. For example, $\mathtt{einsum}((i, i \to \emptyset), \mathrm{a}, \mathrm{b})$ is the inner product of a and b.

LARA can easily express a named version of `einsum` in which α and β are renamings of the key attributes of the input tensors.[1] In fact, it can be shown that LARA can express a *renaming* operator ρ such that $\rho_\alpha \mathsf{A}$ is a tensor with exactly the same data as A but with its key attributes renamed according to α. Thus $\mathtt{einsum}((\alpha, \beta \to \gamma), \mathsf{A}, \mathsf{B})$ can be expressed simply as $\Sigma^{\gamma}_{+} \, (\rho_\alpha \mathsf{A} \bowtie_{\cdot} \rho_\beta \mathsf{B})$. Given that we are dealing with named tensors, a special (and more natural) case is when we use the original attribute names without renaming and, instead of specifying the key attributes in the output, we specify the attributes δ over which we want to sum. In such a case we obtain the expression $\bar{\Sigma}^{\delta}_{+} \, (\mathsf{A} \bowtie_{\cdot} \mathsf{B})$, which resembles the *named tensor contraction* [18, 19].

For the case of MATLANG, since every expression produces a matrix, a vector or a scalar, we can only hope to express `einsum` when $|\gamma| \leq 2$. This is what we prove next.

Theorem 3.6. $\mathtt{einsum}((\alpha, \beta \to \gamma), \mathsf{A}, \mathsf{B})$ *can be expressed in* MATLANG*, provided that* $|\gamma| \leq 2$.

4 CONCLUDING REMARKS

Proposals of declarative query languages for array-based data can be traced back to AQL [14]. With the advent of deep learning, and also the necessity of performing in-database learning due to scalability issues, several new proposals have emerged [7–10, 12]. We picked MATLANG and LARA for our study as they allow to express operations directly over matrices and tensors, and also because they follow a minimalistic presentation that lends itself for theoretical analysis.

Both languages have their pros and cons in terms of expressive power. On the one hand, MATLANG has a simple and intuitive semantics, yet its expressive power is quite limited for expressing practically relevant ML operations. As we explained, this is due to the genericity and locality properties of the language. On the other hand, LARA is more powerful in terms of its capability for expressing standard ML operations. In fact, LARA seems to be a good candidate language for formalizing the *named-tensor* push coming from the deep learning community [18, 19]. Still, several challenging questions regarding the expressive power of LARA remain open: Can LARA express matrix-inversion? If not, can the language express non-local properties? These are interesting questions for future work.

[1]For simplicity we assume that α does not contain repeated attributes, and similarly for β. Repetitions are allowed in general in `einsum`. This case is handled in the full version.

REFERENCES

[1] Robert Brijder, Floris Geerts, Jan Van den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. In *ICDT*, pages 10:1–10:17, 2018.

[2] R. Collobert, K. Kavukcuoglu, and C. Farabet. Torch7: A Matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, 2011.

[3] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[4] Stephan Hoyer, Joe Hamman, and xarray developers. xarray development roadmap. Technical report, 2018. Available at http://xarray.pydata.org/en/stable/roadmap.html, retrieved on March 2019.

[5] Dylan Hutchison, Bill Howe, and Dan Suciu. Lara: A key-value algebra underlying arrays and relations. *CoRR*, abs/1604.03607, 2016.

[6] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of BeyondMR@SIGMOD 2017*, pages 2:1–2:10, 2017.

[7] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. AC/DC: in-database learning thunderstruck. In *Proceedings of the Second Workshop on Data Management for End-To-End Machine Learning, DEEM@SIGMOD 2018, Houston, TX, USA, June 15, 2018*, pages 8:1–8:10, 2018.

[8] Mahmoud Abo Khamis, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. In-database learning with sparse tensors. In *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 325–340, 2018.

[9] Andreas Kunft, Alexander Alexandrov, Asterios Katsifodimos, and Volker Markl. Bridging the gap: towards optimization across linear and relational algebra. In *BeyondMR@SIGMOD 2016, San Francisco, CA, USA, July 1, 2016*, page 1, 2016.

[10] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Tilmann Rabl, and Volker Markl. BlockJoin: Efficient matrix partitioning through joins. *PVLDB*, 10(13):2061–2072, 2017.

[11] Éric Leclercq and Marinette Savonnet. A tensor based data model for polystore: An application to social networks data. In *Proceedings of the 22nd International Database Engineering & Applications Symposium, IDEAS 2018, Villa San Giovanni, Italy, June 18-20, 2018*, pages 110–118, 2018.

[12] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. Mlog: Towards declarative in-database machine learning. *PVLDB*, 10(12):1933–1936, 2017.

[13] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[14] Leonid Libkin, Rona Machlin, and Limsoon Wong. A query language for multidimensional arrays: Design, implementation, and optimization techniques. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996.*, pages 228–239, 1996.

[15] Travis Oliphant. NumPy: A guide to NumPy. USA: Trelgol Publishing, 2006–.

[16] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

[17] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS-W*, 2017.

[18] Alexander M. Rush. Tensor considered harmful. Technical report, Harvard NLP Blog, 2019. Available at http://nlp.seas.harvard.edu/NamedTensor, retrieved on March 2019.

[19] 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.

[20] Florin Rusu and Yu Cheng. A survey on array storage, query languages, and systems. *CoRR*, abs/1302.0103, 2013.

[21] Michael Stonebraker, Paul Brown, Donghui Zhang, and Jacek Becla. Scidb: A database management system for applications with complex analytics. *Computing in Science and Engineering*, 15(3):54–62, 2013.

A PROOFS

Proof of Theorem 3.1

We prove the Theorem by first providing an invariant property for MATLANG expressions.

Let A be a matrix of dimensions $m \times n$, and π_m, π_n permutations over the domains $\{1, \dots, m\}$ and $\{1, \dots, n\}$ respectively [2]. We say that $\pi = (\pi_m, \pi_n)$ is a *matrix permutation*, and its application $\pi(\mathsf{A})$ is such that $\pi(\mathsf{A})_{ij} = \mathsf{A}_{\pi_m(i)\pi_n(j)}$. For every permutation π_n, one can construct a permutation matrix P_{π_n} by permuting the rows of the $(n \times n)$ identity matrix according to π. Permutation matrices are such that for an $(m \times n)$ matrix A it holds that $\pi(\mathsf{A}) = \mathsf{P}_{\pi_n} \cdot \mathsf{A} \cdot \mathsf{P}^T_{\pi_m}$. We will also use the fact that permutation matrices are always orthogonal (i.e. $\mathsf{P}_{\pi_n} \cdot \mathsf{P}^T_{\pi_n} = \mathsf{P}^T_{\pi_n} \cdot \mathsf{P}_{\pi_n} = I$).

We can naturally extend this definition in the following way. Let f be a function that maps every $n \geq 1$ to a permutation $f_n = \pi_n$, . We say that π_f is a matrix permutation, and its application $\pi_f(\mathsf{A})$ to a matrix of dimensions $a \times b$ is such that $\pi_f(\mathsf{A})_{ij} = \mathsf{A}_{f_a(i) f_b(j)}$. From now on, when we say that π is a matrix permutation, we will assume there is an implicit function f defined for it.

We can now state the following invariance property. For every instance $\mathcal{I}$, matrix permutation π and MATLANG expression φ, we have that $\varphi(\pi(\mathcal{I})) = \pi(\varphi(\mathcal{I}))$. We prove this by estructural induction. When $\varphi = A$, the property holds by definition. When $\varphi = \varphi_1 \cdot \varphi_2$, assuming the result of φ_1 has dimensions $a \times b$ and φ_2 has dimensions $b \times c$ (and thus φ has dimensions $a \times c$) we have that:

$$\begin{aligned}
\varphi(\pi(\mathcal{I})) &= \varphi_1(\pi(\mathcal{I})) \cdot \varphi_2(\pi(\mathcal{I})) \\
&= \pi(\varphi_1(\mathcal{I})) \cdot \pi(\varphi_2(\mathcal{I})) \\
&= \mathsf{P}_{\pi_a} \cdot \varphi_1(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_b} \cdot \mathsf{P}_{\pi_b} \cdot \varphi_2(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_c} \\
&= \mathsf{P}_{\pi_a} \cdot \varphi_1(\mathcal{I}) \cdot \varphi_2(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_c} \\
&= \pi(\varphi(\mathcal{I})).
\end{aligned}$$

[2]From now on, we will specify the domain of a permutation just by its subindex

When $\varphi = \varphi_1^T$, assuming the result of φ_1 has dimensions $a \times b$ (and thus φ has dimensions $b \times a$) we have that:

$$\begin{aligned}
\varphi(\pi(\mathcal{I})) &= \varphi_1(\pi(\mathcal{I}))^T \\
&= \pi(\varphi_1(\mathcal{I}))^T \\
&= (\mathsf{P}_{\pi_a} \cdot \varphi_1(\mathcal{I}) \cdot \mathsf{P}_{\pi_b}^T)^T \\
&= \mathsf{P}_{\pi_b} \cdot \varphi_1^{-1}(\mathcal{I}) \cdot \mathsf{P}_{\pi_a}^T \\
&= \pi(\varphi(\mathcal{I})).
\end{aligned}$$

The cases where $\varphi = \mathbf{1}(\varphi_1)$ or $\varphi = diag(\varphi_1)$ are similar.

With our invariance property it is easy to prove Theorem 3.1. Let $\mathcal{S}$ be a schema with two matrices A and B, such that A is of type (α, α) and B of type (β, β). Consider an instance $\mathcal{I}$ that assigns A to the following matrix

$$\mathcal{I}(A) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and B to the matrix

$$\mathcal{I}(B) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Then we have that $(\mathcal{I}(A) * \mathcal{I}(B))$ is given by

$$(\mathcal{I}(A) * \mathcal{I}(B)) = \begin{bmatrix} 2 & 2 & 1 & 0 \\ 2 & 2 & 1 & 0 \\ 1 & 1 & 2 & 1 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

Consider now the permutations $\pi_1 = (1)$, $\pi_3 = (1, 2, 3)$, and $\pi_4 = (1, 3, 2, 4)$. If we consider π a matrix permutation whose implicit function maps to the permutations described above, we have that:

$$\pi(\mathcal{I}(A) * \mathcal{I}(B)) = \begin{bmatrix} 2 & 1 & 2 & 0 \\ 1 & 2 & 1 & 1 \\ 2 & 1 & 2 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$

while

$$\pi(\mathcal{I}(A)) * \pi(\mathcal{I}(B)) = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & 2 & 1 & 1 \\ 0 & 1 & 2 & 2 \\ 0 & 1 & 2 & 2 \end{bmatrix}$$

Thus we have that $\pi(\mathcal{I}(A) * \mathcal{I}(B)) \neq \pi(\mathcal{I}(A)) * \pi(\mathcal{I}(B))$ which is enough to conclude that MATLANG cannot express the convolution.

A.1 Proof of Theorem 3.2

The full LARA language define three operators: Join, Union and Extend. We already defined Join in the body of the paper. For this part we will also need the Extend operator that we next define (for the full definition of Union we refer the reader to [6]).

Let $\mathsf{T}[K, V]$ be an associative table and consider a new set of key and value attributes K' and V'. Now consider a function f that assigns an associative table $\mathsf{A}_{(\bar{k},\bar{v})}[K', V']$ to every tuple $(\bar{k}, \bar{v})$ in T. Then $\mathsf{E} = \text{ext}_f\, \mathsf{T}$ is a new associative table with key attributes $K \cup K'$ and value attributes V' such that $\mathsf{E}(\bar{k} \cup \bar{k}') = \mathsf{A}_{(\bar{k},\bar{v})}(\bar{k}')$.

For the rest of the proof we need to introduce some additional notions. We first define an expression in LARA that computes the cartesian product of two associative tables. Let $\mathsf{A}[K_1, V_1]$ and $\mathsf{B}[K_2, V_2]$ be tables such that $K_1 \cap K_2 = V_1 \cap V_2 = \emptyset$. The cartesian product is a new associative table $\mathsf{C}[K_1, K_2, V_1, V_2]$ such that for every tuple $(\bar{k}_1, \bar{v}_1) \in \mathsf{A}$ and $(\bar{k}_2, \bar{v}_2) \in \mathsf{B}$ we have that $(\bar{k}_1, \bar{k}_2, \bar{v}_1, \bar{v}_2) \in \mathsf{C}$. It is not difficult to prove that the cartesian product is expressible in LARA (see Lemma **??** at the end of this section). Thus we can safely add to the LARA language the operator $\times$ to express the cartesian product between associative tables.

Another operator that we need to simplify our proof is the filter operator. Given an associative table $\mathsf{A}[K, V]$ and a logical expression $\varphi(\bar{x}, \bar{y})$, filtering A with φ, denoted by $\text{filter}_\varphi(\mathsf{A})$ is a new associative table $\mathsf{B}[K, V]$ such that $(\bar{k}, \bar{v})$ is in B if and only if $(\bar{k}, \bar{v}) \in \mathsf{A}$ and $(\bar{k}, \bar{v}) \models \varphi$ (i.e. the logical expression $\varphi(\bar{k}, \bar{v})$ evaluates to *true*). It is not difficult to prove that the filter operator is expressible in LARA (see Lemma **??** at the end of this section). Finally, the renaming operator $\rho_\alpha(\mathsf{A})$ is a simple operator that just change the name of attributes of the associative table A according to the assignment α. It can be easily defined as a special case of ext.

In our proof we use the aggregate operator $+(\cdot)$, that given a multiset of numbers, returns the sum of the multiset. We use two filtering expressions defined as follows:

$$\text{neighbors}(i, i', j, j', m): \quad i' \geq i - \frac{m-1}{2} \;\wedge\; i' \leq i + \frac{m-1}{2} \;\wedge\; j' \geq j - \frac{m-1}{2} \;\wedge\; j' \leq j + \frac{m-1}{2}$$

$$\text{kernel}(i, i', i_k, j, j', j_k, m): \quad i_k = i' - i + \frac{m-1}{2} + 1 \;\wedge\; j_k = j' - j + \frac{m-1}{2} + 1$$

Additionally, we use two ext functions: $\zeta(i, j, v, v', v_k)$, that computes a value attribute v^* such that in every tuple it maps v^* to $v' * v_k$, and $\text{diag}(i, j, v)$, that computes a value attribute m that maps to 1 if $i = j$ and to 0 if not. We can now prove

Theorem 3.2, that is, that LARA can express the 2D matrix convolution.

Let $\mathsf{A}[(i,j),(v)]$ and $\mathsf{K}[(i_k,j_k),(v_k)]$ be two associative tables that represents a matrix and the convolution kernel, respectively. First calculate the dimension of the kernel using

$$\mathsf{M} = \text{⧗}^{\emptyset}_{+(\cdot)} \ \mathrm{ext}_{\mathrm{diag}}\mathsf{K}.$$

Notice $\mathrm{ext}_{\mathrm{diag}}\mathsf{K}$ is a table of sort $[(i,j),(m)]$ and M is a table with no key attributes and a single value attribute m that contains solely the dimension of the kernel. Now we proceed to make the cartesian product of A with itself, K and M For that we need to make a copy of A with renamed attributes. We define the copy of A as $\mathsf{A}' = \rho_{i:i',j:j',v:v'}\mathsf{A}$. The cartesian product is $\mathsf{C} = \mathsf{A} \times \mathsf{A}' \times \mathsf{K} \times \mathsf{M}$. Then

$$\mathsf{C} = \mathrm{filter}_{\mathrm{kernel}}(\mathrm{filter}_{\mathrm{neighbors}}(\mathsf{C})).$$

Now we just multiply v' with v_k and aggregate over i, j obtaining the final result for the convolution with the expression

$$\mathsf{C}^* = \text{⧗}^{i,j}_{+} \ \mathrm{ext}_{\zeta}\mathsf{C}.$$

This table C^* has attributes $[(i,j),(v^*)]$. It is not difficult to see that C^* is a representation of the convolution. Notice that for an arbitrary tuple (a,b,c) in A we have that:

- neighbors will select all the tuples such that they are in a "radius" defined by the size of K (m in this case), with (a,b) at the "center". Notice that neighbors only selects tuples in the table, so theres no need for padding.
- kernel makes a one to one correspondence between the neighborhood and the kernel, such that (a,b) is paired with the "center" of the kernel $((m-1)/2,(m-1)/2)$.

When doing the multiplication of the neighborhood with the kernel and aggregating we obtain that of $\mathsf{C}^*(a,b)$ is exactly the value $(\mathsf{A} * \mathsf{K})_{ab}$. Notice that, since we only use standard operators in Relational Algebra, we can directly translate our query into SQL (see Figure **??**).

We finish this section showing how can one construct a LARA expression for the cartesian product and the filter operator.

Lemma A.1. *The cartesian product is definable in* LARA.

Proof. First consider the following extension functions:

- $\kappa_{\bar{x};\bar{a}}$, the function that, maintaining the original value attributes fixed, adds new key attributes $\bar{x}$ that evaluates to values $\bar{a}$ in each tuple of the extended table, with $|\bar{x}| = |\bar{a}|$.
- $\nu_{\bar{x};\bar{a}}$, the function that, maintaining the original value attributes fixed, adds new value attributes $\bar{x}$ that evaluates to values $\bar{a}$ in each tuple of the extended table, with $|\bar{x}| = |\bar{a}|$.

```
SELECT B.i, B.j, SUM(A.val * K.val) AS val
FROM A, A AS B, K, (
    SELECT COUNT(*) AS m
    FROM K
    WHERE K.i = K.j ) AS KerDim
WHERE
    A.i >= B.i - (m-1)/2  AND  A.i <= B.i + (m-1)/2 AND
    A.j >= B.j - (m-1)/2  AND  A.j <= B.j + (m-1)/2 AND
    K.i = A.i - B.i + (m-1)/2 + 1  AND
    K.j = A.j - B.j + (m-1)/2 + 1
GROUP BY B.i, B.j
```

Figure 1: Convolution $B = A * K$ in SQL.

Given two associative tables $\mathsf{A}[K_A, V_A]$ and $\mathsf{B}[K_B, V_B]$ such that $K_A \cap K_B = \emptyset$ and $V_A \cap V_B = \emptyset$, we define the expression for the cartesian product between them, denoted by $\mathsf{A} \times \mathsf{B}$ as

$$\text{⧗}^{K_A \cup K_B} \ (\mathrm{ext}_{\kappa_{g;1}}\mathrm{ext}_{\nu_{V_B \setminus V_A;\bar{0}}}\mathsf{A}) \bowtie_{+} (\mathrm{ext}_{\kappa_{g;1}}\mathrm{ext}_{\nu_{V_A \setminus V_B;\bar{0}}}\mathsf{B}),$$

with $\bar{0}$ a tuples of 0s of same length that $V_A \setminus V_B$. Notice we used $\text{⧗}^{K_A \cup K_B}$ to eliminate the key attribute g, and is not parametized by an aggregate operator because it does not incur on aggregation. □

Lemma A.2. *The filter operator is definable in* LARA.

Proof. Given an associative table $\mathsf{A}[K,V]$ and a logical expression $\varphi(\bar{x},\bar{y})$, the expression $\mathrm{filter}_{\varphi}(\mathsf{A})$ is equivalent to $\mathrm{ext}_{f_\varphi}\mathsf{A}$, where f_φ is an extension function that returns a table $\mathsf{T}[\emptyset, V]$ such that if $(\bar{k},\bar{v}) \in \mathsf{A}$ and $(\bar{k},\bar{v}) \models \varphi$, then T contains just the values of the tuple, i.e, $\bar{v}$, and if not, T is empty. □

A.2 Proof of Proposition 3.4

We follow the semantics for inverse proposed by Brijder [1], and thus for an expression $\varphi = \mathrm{inv}(\varphi_1)$ and an instance $\mathcal{I}$ we have that $\varphi(\mathcal{I}) = (\varphi_1(\mathcal{I}))^{-1}$ if $\varphi_1(\mathcal{I})$ is an invertible square matrix, and $\varphi(\mathcal{I}) = \mathsf{Z}$ otherwise where Z is a square matrix of the same dimensions of $\varphi_1(\mathcal{I})$ that has a 0 in all its components.

Now to prove the result, we just extend the invariance property presented in the proof of Theorem 3.1 to include the inverse operation. Thus, assume that φ_1 satisfies the invariance property (induction hypothesis) and let $\varphi = \varphi_1^{-1}$. Assuming the result of φ_1 has dimensions $a \times a$ we have that:

$$\begin{aligned}
\varphi(\pi(\mathcal{I})) &= \varphi_1^{-1}(\pi(\mathcal{I})) \\
&= (\varphi_1(\pi(\mathcal{I})))^{-1} \\
&= (\pi(\varphi_1(\mathcal{I})))^{-1} \\
&= (\mathsf{P}_{\pi_a} \cdot \varphi_1(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_a})^{-1} \\
&= (\mathsf{P}^T_{\pi_a})^{-1} \cdot (\varphi_1(\mathcal{I}))^{-1} \cdot (\mathsf{P}_{\pi_a})^{-1} \\
&= \mathsf{P}_{\pi_a} \cdot \varphi_1^{-1}(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_a} \\
&= \mathsf{P}_{\pi_a} \cdot \varphi(\mathcal{I}) \cdot \mathsf{P}^T_{\pi_a} \\
&= \pi(\varphi(\mathcal{I})).
\end{aligned}$$

In Theorem 3.1 we already proved that the convolution operator breaks this invariance property and thus we obtain that convolution cannot be expressed in $\mathsf{MATLANG} + \mathsf{INV}$.

A.3 Einstein summation

We give a formal extension to named tuples for the Einsten summation. The general named Einstein summation expression is $\mathtt{einsum}((\alpha, \beta \to \gamma), \mathsf{A}, \mathsf{B})$, where $\alpha = (k^a_1 : i_1 \dots k^a_n : i_n)$, $\beta = (k^b_1 : j_1 \dots k^b_m : j_m)$ and $\gamma \subseteq \alpha \cup \beta$. The named Einstein summation assigns names to every dimension of the matrices. Notice this gives us a natural link to express named Einstein summations in LARA as this algebra also works on named tuples. In some instances, the Einstein summation assigns the same index to two dimensions. This is interpreted as saying that we only consider on the "diagonal" of such dimensions. For example, $\mathtt{einsum}((k_1 : i\ k_2 : i \to k_1 : i), \mathsf{A})$ calculates the trace of A when A is a matrix. But LARA does not allow an associative table with repeated attribute names. We circumvent this problem in LARA by renaming and applying a filter that selects tuples such their correspond to the diagonal of the dimensions with the same name. Formally, having the expression $\mathtt{einsum}((\alpha, \beta \to \gamma), \mathsf{A}, \mathsf{B})$, for every two dimensions $k^a_i, k^a_j \in \alpha$ with the same name n_{ij}, we construct a filter predicate φ_{ij} that selects only tuples such that $k^a_i = k^a_j$, apply the filter to A, remove k^a_j from α, and then proceed in the same fashion till there are no more repeated indexes. Lets call α' the resulting renaming and A' the resulting tensor. For example, if $\alpha = (k_1 : i, k_2 : i)$, then $\alpha' = (k_1 : i)$ and $\mathsf{A}' = \mathrm{filter}_{k_1 = k_2} \mathsf{A}$. We do the same for repeated names in β and B. Then, finally the einsum expression can be defined simply as $\mathbb{X}^{\gamma}_{+} \, (\rho_{\alpha'} \mathsf{A}' \bowtie_{\cdot} \rho_{\beta'} \mathsf{B}')$.

Proof of Theorem 3.6

As there is a finite number of non-equivalent combinations for α, β and γ, it is enough to provide a MATLANG expression for each of them. We give a few representative examples. We use the binary function $\neq_1 (x, y)$ that is equal to y if $x = 1$ and it is 0 otherwise.

First, we define some shortcuts that will make our expressions simpler.

$$\begin{aligned} \mathtt{rowsum}(\mathsf{A}) &:= \mathsf{A} \cdot \mathtt{1}(\mathsf{A}^T) \\ \mathtt{colsum}(\mathsf{A}) &:= \mathtt{rowsum}(\mathsf{A}^T) \\ \mathtt{sum}(\mathsf{A}) &:= \mathtt{colsum}(\mathtt{rowsum}(\mathsf{A})) \end{aligned}$$

If A is a square matrix, then

$$\mathtt{extdiag}(\mathsf{A}) \ := \ \mathtt{rowsum}(\mathtt{apply}[\neq_1](\mathtt{diag}(\mathtt{1}(\mathsf{A})), \mathsf{A}))$$

With these shortcuts we can more easily define einsum expressions in MATLANG. Below we show some examples (all the other cases are similar). To maintain some uniformity, when $|\gamma| = 1$ we produce always a column vector.

$$\begin{aligned}
\mathtt{einsum}((ij \to i), \mathsf{A}) &= \mathtt{rowsum}(\mathsf{A}) \\
\mathtt{einsum}((ij \to j), \mathsf{A}) &= \mathtt{colsum}(\mathsf{A}) \\
\mathtt{einsum}((ij \to \emptyset), \mathsf{A}) &= \mathtt{sum}(\mathsf{A}) \\
\mathtt{einsum}((ii \to i), \mathsf{A}) &= \mathtt{extdiag}(\mathsf{A}) \\
\mathtt{einsum}((ii, ii \to \emptyset), \mathsf{A}, \mathsf{B}) &= \mathtt{extdiag}(\mathsf{A})^T \cdot \mathtt{extdiag}(\mathsf{B}) \\
\mathtt{einsum}((ij, jk \to ik), \mathsf{A}, \mathsf{B}) &= \mathsf{A} \cdot \mathsf{B} \\
\mathtt{einsum}((ij, ij \to i), \mathsf{A}, \mathsf{B}) &= \mathtt{extdiag}(\mathsf{A} \cdot \mathsf{B}^T) \\
\mathtt{einsum}((ik, kj \to k), \mathsf{A}, \mathsf{B}) &= \mathtt{apply}[\cdot](\mathtt{colsum}(\mathsf{A}), \mathtt{rowsum}(\mathsf{B})) \\
\mathtt{einsum}((ij, kl \to \emptyset), \mathsf{A}, \mathsf{B}) &= \mathtt{sum}(\mathsf{A}) \cdot \mathtt{sum}(\mathsf{B}) \\
\mathtt{einsum}((ij, ik \to \emptyset), \mathsf{A}, \mathsf{B}) &= \mathtt{sum}(\mathsf{A}^T \cdot \mathsf{B})
\end{aligned}$$