On the Expressive Power of Query Languages for Matrices

Robert Brijder
Hasselt University, Hasselt, Belgium

Floris Geerts
University of Antwerp, Antwerp, Belgium

Jan Van den Bussche
Hasselt University, Hasselt, Belgium

Timmy Weerwag
Hasselt University, Hasselt, Belgium

Abstract

We investigate the expressive power of MATLANG, a formal language for matrix manipulation based on common matrix operations and linear algebra. The language can be extended with the operation inv of inverting a matrix. In MATLANG + inv we can compute the transitive closure of directed graphs, whereas we show that this is not possible without inversion. Indeed we show that the basic language can be simulated in the relational algebra with arithmetic operations, grouping, and summation. We also consider an operation eigen for diagonalizing a matrix, which is defined so that different eigenvectors returned for a same eigenvalue are orthogonal. We show that inv can be expressed in MATLANG + eigen. We put forward the open question whether there are boolean queries about matrices, or generic queries about graphs, expressible in MATLANG + eigen but not in MATLANG + inv. The evaluation problem for MATLANG + eigen is shown to be complete for the complexity class $\exists\mathbf{R}$.

2012 ACM Subject Classification Theory of computation → Database theory, Theory of computation → Database query languages (principles), Computing methodologies → Linear algebra algorithms

Keywords and phrases matrix query languages, relational algebra with aggregates, query evaluation problem, graph queries

Digital Object Identifier 10.4230/LIPIcs.ICDT.2018.10

Acknowledgements We thank Bart Kuijpers for telling us about the complexity class $\exists\mathbf{R}$. We thank Lauri Hella and Wied Pakusa for helpful discussions, and Christoph Berkholz and Anuj Dawar for their help with the proof of Proposition 10. R.B. is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

1 Introduction

Data scientists often use matrices to represent their data, as opposed to using the relational data model. These matrices are then manipulated in programming languages such as R or MATLAB. These languages have common operations on matrices built-in, notably matrix multiplication; matrix transposition; elementwise operations on the entries of matrices; solving nonsingular systems of linear equations (matrix inversion); and diagonalization (eigenvalues and eigenvectors). Such programming languages trace back to the APL language [20].

21st International Conference on Database Theory (ICDT 2018).
Editors: Benny Kimelfeld and Yael Amsterdamer; Article No. 10; pp. 10:1–10:17

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Providing database support for matrices and multidimensional arrays has been a long-standing research topic [31], originally geared towards applications in scientific data management, and more recently motivated by machine learning over big data [5, 36, 8, 29].

Database theory and finite model theory provide a rich picture of the expressive power of query languages [1, 24]. In this paper we would like to bring matrix languages into this picture. There is a lot of current interest in languages that combine matrix operations with relational query languages or logics, both in database systems [19] and in finite model theory [10, 11, 18]. In the present study, however, we focus on matrices alone. Indeed, given their popularity, we believe the expressive power of matrix sublanguages also deserves to be understood in its own right.

The contents of this paper can be introduced as follows. We begin the paper by defining the language MATLANG as an analog for matrices of the relational algebra for relations. This language is based on five elementary operations reflecting basic matrix operations available in R, namely, the one-vector; turning a vector in a diagonal matrix; matrix multiplication; matrix transposition; and pointwise function application. We give examples showing that this basic language is capable of expressing common matrix manipulations. For example, the Google matrix of any directed graph G can be computed in MATLANG, starting from the adjacency matrix of G.

Well-typedness and well-definedness notions of MATLANG expressions are captured via a simple data model for matrices. In analogy to the relational model, a schema consists of a number of matrix names, and an instance assigns matrices to the names. Recall that in a relational schema, a relation name is typed by a set of attribute symbols. In our case, a matrix name is typed by a pair $\alpha \times \beta$, where α and β are size symbols that indicate, in a generic manner, the number of rows and columns of the matrix.

In Section 3 we show that our language can be simulated in the relational algebra with aggregates [23, 28], using a standard representation of matrices as relations. The only aggregate function that is needed is summation. In fact, MATLANG is already subsumed by aggregate logic with only three nonnumerical variables. Conversely, MATLANG can express all queries from graph databases (binary relational structures) to binary relations that can be expressed in first-order logic with three variables. In contrast, the four-variable query asking if the graph contains a four-clique, is not expressible.

In Section 4 we extend MATLANG with an operation for inverting a matrix, and we show that the extended language is strictly more expressive. Indeed, the transitive closure of binary relations becomes expressible. The possibility of reducing transitive closure to matrix inversion has been pointed out by several researchers [26, 9, 33]. We show that the restricted setting of MATLANG suffices for this reduction to work. That transitive closure is not expressible without inversion, follows from the locality of relational algebra with aggregates [28].

Another prominent operation of linear algebra, with many applications in data mining and graph analysis [16, 27], is to return eigenvectors and eigenvalues. There are various ways to define this operator formally. In Section 5 we define the operation eigen to return a basis of eigenvectors, in which eigenvectors for a same eigenvalue are orthogonal. We show that the resulting language MATLANG + eigen can express inversion. The argument is well known from linear algebra, but our result shows that it can be carried out in MATLANG, once more attesting that we have defined an adequate matrix language. It is natural to conjecture that MATLANG + eigen is actually strictly more powerful than MATLANG + inv in expressing, say, boolean queries about matrices. Proving this is an interesting open problem.

Finally, in Section 6 we look into the evaluation problem for MATLANG+eigen expressions. In practice, matrix computations are performed using techniques from numerical mathematics [14]. It remains of foundational interest, however, to know whether the evaluation of expressions is effectively computable. We need to define this problem with some care, since we work with arbitrary complex numbers. Even if the inputs are, say, 0-1 matrices, the outputs of the eigen operation can be complex numbers. Moreover, until now we have allowed arbitrary pointwise functions, which we should restrict somehow if we want to discuss computability. Our approach is to restrict pointwise functions to be semi-algebraic, i.e., definable over the real numbers. We will observe that the input-output relation of an expression e, applied to input matrices of given dimensions, is definable in the existential theory of the real numbers, by a formula of size polynomial in the size of e and the given dimensions. This places natural decision versions of the evaluation problem for MATLANG + eigen in the complexity class $\exists\mathbf{R}$ (combined complexity). We show moreover that there exists a fixed expression (data complexity) for which the evaluation problem is $\exists\mathbf{R}$-complete, even restricted to input matrices with integer entries. It also follows that equivalence of expressions, over inputs of given dimensions, is decidable.

2 MATLANG

We assume a sufficient supply of *matrix variables*, which serve to indicate the inputs to expressions in MATLANG. Variables can also be introduced in let-constructs inside expressions. The syntax of MATLANG expressions is defined by the grammar:

$e ::= M$		(matrix variable)
	$\mathsf{let}\ M = e_1\ \mathsf{in}\ e_2$	(local binding)
|	e^*	(conjugate transpose)
|	$\mathbf{1}(e)$	(one-vector)
|	$\mathsf{diag}(e)$	(diagonalization of a vector)
|	$e_1 \cdot e_2$	(matrix multiplication)
|	$\mathsf{apply}[f](e_1, \ldots, e_n)$	(pointwise application, $f \in \Omega$)

In the last rule, f is the name of a function $f : \mathbf{C}^n \to \mathbf{C}$, where $\mathbf{C}$ denotes the complex numbers. Formally, the syntax of MATLANG is parameterized by a repertoire Ω of such functions, but for simplicity we will not reflect this in the notation.

▶ **Example 1.** Let $c \in \mathbf{C}$ be a constant; we also use c as a name for the constant function $c : \mathbf{C} \to \mathbf{C} : z \mapsto c$. Then

$$\mathsf{let}\ N = \mathbf{1}(M)^*\ \mathsf{in}\ \ \mathsf{apply}[c](\mathbf{1}(N))$$

is an example of an expression. At this point, this is a purely syntactical example; we will see its semantics shortly. The expression is actually equivalent to $\mathsf{apply}[c](\mathbf{1}(\mathbf{1}(M)^*))$. The let-construct is useful to give names to intermediate results, but is not essential for now. It will become essential later, when we enrich MATLANG with the eigen operation. ◀

In defining the semantics of the language, we begin by defining the basic matrix operations. Following practical matrix sublanguages such as R or MATLAB, we will work throughout with matrices over the complex numbers. However, a real-number version of the language could be defined as well.

$$\begin{pmatrix} 0 & 1+i \\ 2 & 3-i \\ 4+4i & 5 \end{pmatrix}^* = \begin{pmatrix} 0 & 2 & 4-4i \\ 1-i & 3+i & 5 \end{pmatrix} \qquad \mathbf{1}\begin{pmatrix} 2 & 3 & 4 \\ 4 & 5 & 6 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \cdot \begin{pmatrix} 6 & 5 & 4 & 3 \\ 2 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 10 & 7 & 4 & 1 \\ 26 & 19 & 12 & 5 \\ 42 & 31 & 20 & 9 \end{pmatrix} \qquad \mathsf{diag}\begin{pmatrix} 6 \\ 7 \end{pmatrix} = \begin{pmatrix} 6 & 0 \\ 0 & 7 \end{pmatrix}$$

$$\mathsf{apply}[\dot{-}]\left(\begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}\right) = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Figure 1 Basic matrix operations of MATLANG. The matrix multiplication example is taken from Axler's book [3].

Transpose: If A is a matrix then A^* is its conjugate transpose. So, if A is an $m \times n$ matrix then A^* is an $n \times m$ matrix and the entry $A^*_{i,j}$ is the complex conjugate of the entry $A_{j,i}$.

One-vector: If A is an $m \times n$ matrix then $\mathbf{1}(A)$ is the $m \times 1$ column vector consisting of all ones.

Diag: If v is an $m \times 1$ column vector then $\mathsf{diag}(v)$ is the $m \times m$ diagonal square matrix with v on the diagonal and zero everywhere else.

Matrix multiplication: If A is an $m \times n$ matrix and B is an $n \times p$ matrix then the well known matrix multiplication AB is defined to be the $m \times p$ matrix where $(AB)_{i,j} = \sum_{k=1}^{n} A_{i,k}B_{k,j}$. In MATLANG we explicitly denote this as $A \cdot B$.

Pointwise application: If $A^{(1)}, \ldots, A^{(n)}$ are matrices of the same dimensions $m \times p$, then $\mathsf{apply}[f](A^{(1)}, \ldots, A^{(n)})$ is the $m \times p$ matrix C where $C_{i,j} = f(A^{(1)}_{i,j}, \ldots, A^{(n)}_{i,j})$.

▶ **Example 2.** The operations are illustrated in Figure 1. In the pointwise application example, we use the function $\dot{-}$ defined by $x \dot{-} y = x - y$ if x and y are both real numbers and $x \geq y$, and $x \dot{-} y = 0$ otherwise.

2.1 Formal semantics

The formal semantics of expressions is defined in a straightforward manner, as shown in Figure 2. An *instance* I is a function, defined on a nonempty finite set $\mathrm{var}(I)$ of matrix variables, that assigns a matrix to each element of $\mathrm{var}(I)$. Figure 2 provides the rules that allow to derive that an expression e, on an instance I, successfully evaluates to a matrix A. We denote this success by $e(I) = A$. The reason why an evaluation may not succeed can be found in the rules that have a condition attached to them. The rule for variables fails when an instance simply does not provide a value for some input variable. The rules for diag, apply, and matrix multiplication have conditions on the dimensions of matrices, that need to be satisfied for the operations to be well-defined.

▶ **Example 3** (Scalars)**.** The expression from Example 1, regardless of the matrix assigned to M, evaluates to the 1×1 matrix whose single entry equals c. We introduce the shorthand c for this constant expression. Obviously, in practice, scalars would be built in the language and would not be computed in such a roundabout manner. In this paper, however, we are interested in expressiveness, so we start from a minimal language and then see what is already expressible in this language.

$$\frac{M \in \mathrm{var}(I)}{M(I) = I(M)} \qquad \frac{e_1(I) = A \qquad e_2(I[M := A]) = B}{(\mathsf{let}\ M = e_1\ \mathsf{in}\ e_2)(I) = B} \qquad \frac{e(I) = A}{e^*(I) = A^*} \qquad \frac{e(I) = A}{\mathbf{1}(e)(I) = \mathbf{1}(A)}$$

$$\frac{e(I) = A \qquad A \text{ is a column vector}}{\mathsf{diag}(e)(I) = \mathsf{diag}(A)}$$

$$\frac{e_1(I) = A \qquad e_2(I) = B \qquad \text{number of columns of } A \text{ equals the number of rows of } B}{e_1 \cdot e_2(I) = A \cdot B}$$

$$\frac{\forall k = 1, \ldots, n : (e_k(I) = A_k) \qquad \text{all } A_k \text{ have the same dimensions}}{\mathsf{apply}[f](e_1, \ldots, e_n)(I) = \mathsf{apply}[f](A_1, \ldots, A_n)}$$

Figure 2 Big-step operational semantics of MATLANG. The notation $I[M := A]$ denotes the instance that is equal to I, except that M is mapped to the matrix A.

▶ **Example 4** (Scalar multiplication)**.** Let A be any matrix and let C be a 1×1 matrix; let c be the value of C's single entry. Viewing C as a scalar, we define the operation $C \odot A$ as multiplying every entry of A by c. We can express $C \odot A$ as

$$\mathsf{let}\ M = \mathbf{1}(A) \cdot C \cdot \mathbf{1}(A^*)^*\ \mathsf{in}\ \mathsf{apply}[\times](M, A).$$

If A is an $m \times n$ matrix, we compute in variable M the $m \times n$ matrix where every entry equals c. Then pointwise multiplication is used to do the scalar multiplication.

▶ **Example 5** (Google matrix)**.** Let A be the adjacency matrix of a directed graph (modeling the Web graph) on n nodes numbered $1, \ldots, n$. Let $0 < d < 1$ be a fixed "damping factor". Let k_i denote the outdegree of node i. For simplicity, we assume k_i is nonzero for every i. Then the Google matrix [7, 6] of A is the $n \times n$ matrix G defined by

$$G_{i,j} = d\frac{A_{ij}}{k_i} + \frac{1-d}{n}.$$

The calculation of G from A can be expressed in MATLANG as follows:

$$\begin{array}{l}\mathsf{let}\ J = \mathbf{1}(A) \cdot \mathbf{1}(A)^*\ \mathsf{in}\\ \mathsf{let}\ K = A \cdot J\ \mathsf{in}\\ \mathsf{let}\ B = \mathsf{apply}[/](A, K)\ \mathsf{in}\\ \mathsf{let}\ N = \mathbf{1}(A)^* \cdot \mathbf{1}(A)\ \mathsf{in}\\ \mathsf{apply}[+](d \odot B, (1-d) \odot \mathsf{apply}[1/x](N) \odot J)\end{array}$$

In variable J we compute the $n \times n$ matrix where every entry equals one. In K we compute the $n \times n$ matrix where all entries in the ith row equal k_i. In N we compute the 1×1 matrix containing the value n. The pointwise functions applied are addition, division, and reciprocal. We use the shorthand for constants (d and $1 - d$) from Example 3, and the shorthand $\odot$ for scalar multiplication from Example 4.

▶ **Example 6** (Minimum of a vector). Let $v = (v_1, \ldots, v_n)^*$ be a column vector of real numbers; we would like to extract the minimum from v. This can be done as follows:

let $V = v \cdot \mathbf{1}(v)^*$ in
let $C = \mathsf{apply}[\leq](V, V^*) \cdot \mathbf{1}(v)$ in
let $N = \mathbf{1}(v)^* \cdot \mathbf{1}(v)$ in
let $S = \mathsf{apply}[=](C, \mathbf{1}(v) \cdot N)$ in
let $M = \mathsf{apply}[1/x](S^* \cdot \mathbf{1}(v))$ in
$M \cdot v^* \cdot S$

The pointwise functions applied are $\leq$, which returns 1 on (x, y) if $x \leq y$ and 0 otherwise; $=$, defined analogously; and the reciprocal function. In variable V we compute a square matrix holding n copies of v. Then in variable C we compute the $n \times 1$ column vector where C_i counts the number of v_j such that $v_i \leq v_j$. If $C_i = n$ then v_i equals the minimum. Variable N computes the scalar n and column vector S is a selector where $S_i = 1$ if v_i equals the minimum, and $S_i = 0$ otherwise. Since the minimum may appear multiple times in v, we compute in M the inverse of the multiplicity. Finally we sum the different occurrences of the minimum in v and divide by the multiplicity.

2.2 Types and schemas

We have already remarked that, due to conditions on the dimensions of matrices, MATLANG expressions are not well-defined on all instances. For example, if I is an instance where $I(M)$ is a 3×4 matrix and $I(N)$ is a 2×4 matrix, then the expression $M \cdot N$ is not defined on I. The expression $M \cdot N^*$, however, is well-defined on I. We now introduce a notion of schema, which assigns types to matrix names, so that expressions can be type-checked against schemas.

Our types need to be able to guarantee equalities between numbers of rows or numbers of columns, so that apply and matrix multiplication can be typechecked. Our types also need to be able to recognize vectors, so that diag can be typechecked.

Formally, we assume a sufficient supply of *size symbols*, which we will denote by the letters α, β, γ. A size symbol represents the number of rows or columns of a matrix. Together with an explicit 1, we can indicate arbitrary matrices as $\alpha \times \beta$, square matrices as $\alpha \times \alpha$, column vectors as $\alpha \times 1$, row vectors as $1 \times \alpha$, and scalars as 1×1. Formally, a *size term* is either a size symbol or an explicit 1. A *type* is then an expression of the form $s_1 \times s_2$ where s_1 and s_2 are size terms. Finally, a *schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathrm{var}(\mathcal{S})$ of matrix variables, that assigns a type to each element of $\mathrm{var}(\mathcal{S})$.

The typechecking of expressions is now shown in Figure 3. The figure provides the rules that allow to infer an output type τ for an expression e over a schema $\mathcal{S}$. To indicate that a type can be successfully inferred, we use the notation $\mathcal{S} \vdash e : \tau$. When we cannot infer a type, we say e is not well-typed over $\mathcal{S}$. For example, when $\mathcal{S}(M) = \alpha \times \beta$ and $\mathcal{S}(N) = \gamma \times \beta$, then the expression $M \cdot N$ is not well-typed over $\mathcal{S}$. The expression $M \cdot N^*$, however, is well-typed with output type $\alpha \times \gamma$.

To establish the soundness of the type system, we need a notion of conformance of an instance to a schema.

Formally, a *size assignment* σ is a function from size symbols to positive natural numbers. We extend σ to any size term by setting $\sigma(1) = 1$. Now, let $\mathcal{S}$ be a schema and I an instance with $\mathrm{var}(I) = \mathrm{var}(\mathcal{S})$. We say that I is an instance of $\mathcal{S}$ if there is a size assignment σ such that for all $M \in \mathrm{var}(\mathcal{S})$, if $\mathcal{S}(M) = s_1 \times s_2$, then $I(M)$ is a $\sigma(s_1) \times \sigma(s_2)$ matrix. In that case we also say that I *conforms* to $\mathcal{S}$ by the size assignment σ.

$$\frac{M \in \mathrm{var}(\mathcal{S})}{\mathcal{S} \vdash M : \mathcal{S}(M)} \qquad \frac{\mathcal{S} \vdash e_1 : \tau_1 \qquad \mathcal{S}[M := \tau_1] \vdash e_2 : \tau_2}{\mathcal{S} \vdash \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2 : \tau_2} \qquad \frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash e^* : s_2 \times s_1}$$

$$\frac{\mathcal{S} \vdash e : s_1 \times s_2}{\mathcal{S} \vdash \mathbf{1}(e) : s_1 \times 1} \qquad \frac{\mathcal{S} \vdash e : s \times 1}{\mathcal{S} \vdash \mathsf{diag}(e) : s \times s} \qquad \frac{\mathcal{S} \vdash e_1 : s_1 \times s_2 \qquad \mathcal{S} \vdash e_2 : s_2 \times s_3}{\mathcal{S} \vdash e_1 \cdot e_2 : s_1 \times s_3}$$

$$\frac{n > 0 \qquad f : \mathbf{C}^n \to \mathbf{C} \qquad \forall k = 1, \ldots, n : (\mathcal{S} \vdash e_k : \tau)}{\mathcal{S} \vdash \mathsf{apply}[f](e_1, \ldots, e_n) : \tau}$$

Figure 3 Typechecking MATLANG. The notation $\mathcal{S}[M := \tau]$ denotes the schema that is equal to $\mathcal{S}$, except that M is mapped to the type τ.

We now obtain the following obvious but desirable property.

▶ **Proposition 7** (Safety). *If $\mathcal{S} \vdash e : s_1 \times s_2$, then for every instance I conforming to $\mathcal{S}$, by size assignment σ, the matrix $e(I)$ is well-defined and has dimensions $\sigma(s_1) \times \sigma(s_2)$.*

3 Expressive power of MATLANG

It is natural to represent an $m \times n$ matrix A by a ternary relation

$$Rel_2(A) := \{(i, j, A_{i,j}) \mid i \in \{1, \ldots, m\},\ j \in \{1, \ldots, n\}\}.$$

In the special case where A is an $m \times 1$ matrix (column vector), A can also be represented by a binary relation $Rel_1(A) := \{(i, A_{i,1}) \mid i \in \{1, \ldots, m\}\}$. Similarly, a $1 \times n$ matrix (row vector) A can be represented by $Rel_1(A) := \{(j, A_{1,j}) \mid j \in \{1, \ldots, n\}\}$. Finally, a 1×1 matrix (scalar) A can be represented by the unary singleton relation $Rel_0(A) := \{(A_{1,1})\}$.

Note that in MATLANG, we perform calculations on matrix entries, but not on row or column indices. This fits well to the relational model with aggregates as formalized by Libkin [28]. In this model, the columns of relations are typed as "base", indicated by **b**, or "numerical", indicated by **n**. In the relational representations of matrices presented above, the last column is of type **n** and the other columns (if any) are of type **b**. In particular, in our setting, numerical columns hold complex numbers.

Given this representation of matrices by relations, MATLANG can be simulated in the relational algebra with aggregates. Actually, the only aggregate operation we need is summation. We will not reproduce the formal definition of the relational algebra with summation [28], but note the following salient points:

- Expressions are built up from relation names using the classical operations union, set difference, cartesian product ($\times$), selection (σ), and projection (π), plus two new operations: *function application* and *summation*.
- For selection, we only use equality and nonequality comparisons on base columns. No selection on numerical columns will be needed in our setting.
- For any function $f : \mathbf{C}^n \to \mathbf{C}$, the operation $\mathsf{apply}[f; i_1, \ldots, i_n]$ can be applied to any relation r having columns $i_1, \ldots, i_n$, which must be numerical. The result is the relation $\{(t, f(t(i_1), \ldots, t(i_n))) \mid t \in r\}$, appending a numerical column to r. We allow $n = 0$, in which case f is a constant.

- The operation $\mathsf{sum}[i; i_1, \ldots, i_n]$ can be applied to any relation r having columns i, i_1, …, i_n, where column i must be numerical. In our setting we only need the operation in cases where columns i_1, …, i_n are base columns. The result of the operation is the relation

$$\{(t(i_1), \ldots, t(i_n), \sum_{t' \in \mathsf{group}[i_1,\ldots,i_n](r,t)} t'(i)) \mid t \in r\},$$

where

$$\mathsf{group}[i_1, \ldots, i_n](r, t) = \{t' \in r \mid t'(i_1) = t(i_1) \wedge \cdots \wedge t'(i_n) = t(i_n)\}.$$

Again, n can be zero, in which case the result is a singleton.

3.1 From MATLANG to relational algebra with summation

To state the translation formally, we assume a supply of *relation variables*, which, for convenience, we can take to be the same as the matrix variables. A *relation type* is a tuple of **b**'s and **n**'s. A *relational schema* $\mathcal{S}$ is a function, defined on a nonempty finite set $\mathrm{var}(\mathcal{S})$ of relation variables, that assigns a relation type to each element of $\mathrm{var}(\mathcal{S})$.

One can define well-typedness for expressions in the relation algebra with summation, and define the output type. We omit this definition here, as it follows a well-known methodology [37] and is analogous to what we have already done for MATLANG in Section 2.2.

To define relational instances, we assume a countably infinite universe **dom** of abstract atomic data elements. It is convenient to assume that the natural numbers are contained in **dom**. We stress that this assumption is not essential but simplifies the presentation. Alternatively, we would have to work with explicit embeddings from the natural numbers into **dom**.

Let τ be a relation type. A *tuple of type* τ is a tuple $(t(1), \ldots, t(n))$ of the same arity as τ, such that $t(i) \in \mathbf{dom}$ when $\tau(i) = \mathbf{b}$, and $t(i)$ is a complex number when $\tau(i) = \mathbf{n}$. A *relation of type* τ is a finite set of tuples of type τ. An *instance* of a relational schema $\mathcal{S}$ is a function I defined on $\mathrm{var}(\mathcal{S})$ so that $I(R)$ is a relation of type $\mathcal{S}(R)$ for every $R \in \mathrm{var}(\mathcal{S})$.

We must connect the matrix data model to the relational data model. Let $\tau = s_1 \times s_2$ be a matrix type. Let us call τ a *general type* if s_1 and s_2 are both size symbols; a *vector type* if s_1 is a size symbol and s_2 is 1, or vice versa; and the *scalar type* if τ is 1×1. To every matrix type τ we associate a relation type

$$Rel(\tau) := \begin{cases} (\mathbf{b}, \mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is general;} \\ (\mathbf{b}, \mathbf{n}) & \text{if } \tau \text{ is a vector type;} \\ (\mathbf{n}) & \text{if } \tau \text{ is scalar.} \end{cases}$$

Then to every matrix schema $\mathcal{S}$ we associate the relational schema $Rel(\mathcal{S})$ where $Rel(\mathcal{S})(M) = Rel(\mathcal{S}(M))$ for every $M \in \mathrm{var}(\mathcal{S})$. For each instance I of $\mathcal{S}$, we define the instance $Rel(I)$ over $Rel(\mathcal{S})$ by

$$Rel(I)(M) = \begin{cases} Rel_2(I(M)) & \text{if } \mathcal{S}(M) \text{ is a general type;} \\ Rel_1(I(M)) & \text{if } \mathcal{S}(M) \text{ is a vector type;} \\ Rel_0(I(M)) & \text{if } \mathcal{S}(M) \text{ is the scalar type.} \end{cases}$$

Here we use the relational representations Rel_2, Rel_1 and Rel_0 of matrices introduced in the beginning of Section 3.

▶ **Theorem 8.** *Let $\mathcal{S}$ be a matrix schema, and let e be a* MATLANG *expression that is well-typed over $\mathcal{S}$ with output type τ. Let $\ell = 2$, 1, or 0, depending on whether τ is general, a vector type, or scalar, respectively.*

1. *There exists an expression $Rel(e)$ in the relational algebra with summation, that is well-typed over $Rel(\mathcal{S})$ with output type $Rel(\tau)$, such that for every instance I of $\mathcal{S}$, we have $Rel_\ell(e(I)) = Rel(e)(Rel(I))$.*
2. *The expression $Rel(e)$ uses neither set difference, nor selection conditions on numerical columns.*
3. *The only functions used in $Rel(e)$ are those used in pointwise applications in e; complex conjugation; multiplication of two numbers; and the constant functions 0 and 1.*

Proof. We only give a few representative examples.

- If M is of type $\alpha \times \beta$ then $Rel(M^*)$ is $\mathsf{apply}[\overline{z}; 3]\, \pi_{2,1,3}(M)$, where $\overline{z}$ is the complex conjugate. If M is of type $\alpha \times 1$, however, $Rel(M^*)$ is $\mathsf{apply}[\overline{z}; 2](M)$.
- If M is of type $1 \times \alpha$ then $Rel(\mathbf{1}(M))$ is $\pi_3(\mathsf{apply}[1; 2](M))$. Here, 1 stands for the constant 1 function.
- If M is of type $\alpha \times 1$ then $Rel(\mathsf{diag}(M))$ is

$$\sigma_{\$1=\$2}(\pi_1(M) \times M) \cup \mathsf{apply}[0; \,]\, \sigma_{\$1 \neq \$2}(\pi_1(M) \times \pi_1(M)).$$

- If M is of type $\alpha \times \beta$ and N is of type $\beta \times \gamma$, then $Rel(M \cdot N)$ is

$$\mathsf{sum}[7; 1, 5]\ \mathsf{apply}[\times; 3, 6]\, \sigma_{\$2=\$4}(M \times N).$$

  If, however, M is of type $\alpha \times 1$ and N is of type 1×1, then $Rel(M \cdot N)$ is

$$\pi_{1,4}\ \mathsf{apply}[\times; 2, 3](M \times N).$$

  We use pointwise multiplication.
- If M and N are of type $1 \times \beta$ then $Rel(\mathsf{apply}[f](M, N))$ is $\pi_{1,5}\ \mathsf{apply}[f; 2, 4]\, \sigma_{\$1=\$3}(M \times N)$.

We may ignore the let-construct as it does not add expressive power. ◀

▶ Remark. The different treatment of general types, vector types, and scalar types is necessary because in our version of the relational algebra, selections can only compare base columns for equality; in particular we can not select for the value 1.

▶ Remark. We can sharpen the above theorem a bit if we work in the relational calculus with aggregates. Every MATLANG expression can already be expressed by a formula in the relational calculus with summation that uses only three distinct base variables (variables ranging over values in base columns).

3.2 Expressing graph queries

So far we have looked at expressing matrix queries in terms of relational queries. It is also natural to express relational queries as matrix queries. This works best for binary relations, or graphs, which we can represent by their adjacency matrices.

Formally, define a *graph schema* to be a relational schema where every relation variable is assigned the type $(\mathbf{b}, \mathbf{b})$ of arity two. We define a *graph instance* as an instance I of a graph schema, where the active domain of I equals $\{1, \ldots, n\}$ for some positive natural number n. The assumption that the active domain always equals an initial segment of the natural numbers is convenient for forming the bridge to matrices. This assumption, however, is not essential for our results to hold. Indeed, the logics we consider do not have any built-in

predicates on base variables, besides equality. Hence, they view the active domain elements as abstract data values.

To every graph schema $\mathcal{S}$ we associate a matrix schema $Mat(\mathcal{S})$, where $Mat(\mathcal{S})(R) = \alpha \times \alpha$ for every $R \in \text{var}(\mathcal{S})$, for a fixed size symbol α. So, all matrices are square matrices of the same dimension. Let I be a graph instance of $\mathcal{S}$, with active domain $\{1, \ldots, n\}$. We will denote the $n \times n$ adjacency matrix of a binary relation r over $\{1, \ldots, n\}$ by $Adj_I(r)$. Now any such instance I is represented by the matrix instance $Mat(I)$ over $Mat(\mathcal{S})$, where $Mat(I)(R) = Adj_I(I(R))$ for every $R \in \text{var}(\mathcal{S})$.

A *graph query* over a graph schema $\mathcal{S}$ is a function that maps each graph instance I of $\mathcal{S}$ to a binary relation on the active domain of I. We say that a MATLANG expression e *expresses* the graph query q if e is well-typed over $Mat(\mathcal{S})$ with output type $\alpha \times \alpha$, and for every graph instance I of $\mathcal{S}$, we have $Adj_I(q(I)) = e(Mat(I))$.

We can now give a partial converse to Theorem 8. We assume active-domain semantics for first-order logic [1]. Please note that the following result deals only with pure first-order logic, without aggregates or numerical columns.

▶ **Theorem 9.** *Every graph query expressible in* FO^3 *(first-order logic with equality, using at most three distinct variables) is expressible in* MATLANG. *The only functions needed in pointwise applications are boolean functions on* $\{0, 1\}$*, and testing if a number is positive.*

We can complement the above theorem by showing that the quintessential first-order query requiring four variables is not expressible.

▶ **Proposition 10.** *The graph query over a single binary relation R that maps I to $I(R)$ if $I(R)$ contains a four-clique, and to the empty relation otherwise, is not expressible in* MATLANG.

4 Matrix inversion

Matrix inversion (solving nonsingular systems of linear equations) is an ubiquitous operation in data analysis. We can extend MATLANG with matrix inversion as follows. Let $\mathcal{S}$ be a schema and e be an expression that is well-typed over $\mathcal{S}$, with output type of the form $\alpha \times \alpha$. Then the expression e^{-1} is also well-typed over $\mathcal{S}$, with the same output type $\alpha \times \alpha$. The semantics is defined as follows. For an instance I, if $e(I)$ is an invertible matrix, then $e^{-1}(I)$ is defined to be the inverse of $e(I)$; otherwise, it is defined to be the zero square matrix of the same dimensions as $e(I)$. The extension of MATLANG with inversion is denoted by MATLANG + inv.

▶ **Example 11** (PageRank)**.** Recall Example 5 where we computed the Google matrix of A. In the process we already showed how to compute the $n \times n$ matrix B defined by $B_{i,j} = A_{i,j}/k_i$, and the scalar N holding the value n. So, in the following expression, we assume we already have B and N. Let I be the $n \times n$ identity matrix, and let $\mathbf{1}$ denote the $n \times 1$ column vector consisting of all ones. The PageRank vector v of A can be computed as follows [12]:

$$v = \frac{1-d}{n}(I - dB)^{-1}\mathbf{1}.$$

This calculation is readily expressed in MATLANG + inv as

$$(1-d) \odot \mathsf{apply}[1/x](N) \odot \mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(A)), d \odot B)^{-1} \cdot \mathbf{1}(A).$$

▶ **Example 12** (Transitive closure). We next show that the reflexive-transitive closure of a binary relation is expressible in MATLANG + inv. Let A be the adjacency matrix of a binary relation r on $\{1,\ldots,n\}$. Let I be the $n \times n$ identity matrix, expressible as $\mathsf{diag}(\mathbf{1}(A))$. From earlier examples we know how to compute the scalar 1×1 matrix N holding the value n. The matrix $B = \frac{1}{n+1}A$ has 1-norm strictly less than 1, so $S = \sum_{k=0}^{\infty} B^k$ converges, and is equal to $(I - B)^{-1}$ [14, Lemma 2.3.3]. Now (i, j) belongs to the reflexive-transitive closure of r if and only if $S_{i,j}$ is nonzero. Thus, we can express the reflexive-transitive closure of r as

$$\mathsf{apply}[\neq 0]\big(\mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(A)), \mathsf{apply}[1/(x+1)](N) \odot A)^{-1}\big),$$

where $x \neq 0$ is 1 if $x \neq 0$ and 0 otherwise. We can obtain the transitive closure by multiplying the above expression with A. ◀

By Theorem 8, any graph query expressible in MATLANG is expressible in the relational algebra with aggregates. It is known [17, 28] that such queries are local. The transitive-closure query from Example 12, however, is not local. We thus conclude:

▶ **Theorem 13.** MATLANG + inv *is strictly more powerful than* MATLANG *in expressing graph queries.*

Once we have the transitive closure, we can do many other things such as checking bipartiteness of undirected graphs, checking connectivity, checking cyclicity. MATLANG is expressive enough to reduce these queries to the transitive-closure query, as shown in the following example for bipartiteness. The same approach via FO^3 can be used for connectedness or cyclicity.

▶ **Example 14** (Bipartiteness). To check bipartiteness of an undirected graph, given as a symmetric binary relation R without self-loops, we first compute the transitive closure T of the composition of R with itself. Then the FO^3 condition $\neg\exists x\exists y(R(x, y) \wedge T(y, x))$ expresses that R is bipartite (no odd cycles). The result now follows from Theorem 9.

▶ **Example 15** (Number of connected components). Using transitive closure we can also easily compute the number of connected components of a binary relation R on $\{1,\ldots,n\}$, given as an adjacency matrix. We start from the union of R and its converse. This union, denoted by S, is expressible by Theorem 9. We then compute the reflexive-transitive closure C of S. Now the number of connected components of R equals $\sum_{i=1}^{n} 1/k_i$, where k_i is the degree of node i in C. This sum is simply expressible as $\mathbf{1}(C)^* \cdot \mathsf{apply}[1/x](C \cdot \mathbf{1}(C))$.

5 Eigenvalues

Another workhorse in data analysis is diagonalizing a matrix, i.e., finding a basis of eigenvectors. Formally, we define the operation eigen as follows. Let A be an $n \times n$ matrix. Recall that A is called diagonalizable if there exists a basis of $\mathbf{C}^n$ consisting of eigenvectors of A. In that case, there also exists such a basis where eigenvectors corresponding to a same eigenvalue are orthogonal. Accordingly, we define $\mathsf{eigen}(A)$ to return an $n \times n$ matrix, the columns of which form a basis of $\mathbf{C}^n$ consisting of eigenvectors of A, where eigenvectors corresponding to a same eigenvalue are orthogonal. If A is not diagonalizable, we define $\mathsf{eigen}(A)$ to be the $n \times n$ zero matrix.

Note that eigen is nondeterministic; in principle there are infinitely many possible results. This models the situation in practice where numerical packages such as R or MATLAB return approximations to the eigenvalues and a set of corresponding eigenvectors. Eigenvectors,

however, are not unique. Hence, some care must be taken in extending MATLANG with the eigen operator. Syntactically, as for inversion, whenever e is a well-typed expression with a square output type, we now also allow the expression $\mathsf{eigen}(e)$, with the same output type. Semantically, however, the rules of Figure 2 must be adapted so that they do not infer statements of the form $e(I) = B$, but rather of the form $B \in e(I)$, i.e., B is a possible result of $e(I)$. The let-construct now becomes crucial; it allows us to assign a possible result of eigen to a new variable, and work with that intermediate result consistently.

In this and the next section, we assume notions from linear algebra. An excellent introduction to the subject has been given by Axler [3].

▶ Remark (Eigenvalues). We can easily recover the eigenvalues from the eigenvectors, using inversion. Indeed, if A is diagonalizable and $B \in \mathsf{eigen}(A)$, then $\Lambda = B^{-1}AB$ is a diagonal matrix with all eigenvalues of A on the diagonal, so that the ith eigenvector in B corresponds to the eigenvalue in the ith column of Λ. This is the well-known eigendecomposition. However, the same can also be accomplished without using inversion. Indeed, suppose $B = (v_1, \ldots, v_n)$, and let λ_i be the eigenvalue to which v_i corresponds. Then $AB = (\lambda_1 v_1, \ldots, \lambda_n v_n)$. Each eigenvector is nonzero, so we can divide away the entries from B in AB (setting division by zero to zero). We thus obtain a matrix where the ith column consists of zeros or λ_i, with at least one occurrence of λ_i. By counting multiplicities, dividing them out, and finally summing, we obtain $\lambda_1, \ldots, \lambda_n$ in a column vector. We can apply a final diag to get it back into diagonal form. The MATLANG expression for doing all this uses similar tricks as those shown in Examples 5 and 6. ◀

The above remark suggests a shorthand in MATLANG + eigen where we return both B and Λ together:

$$\mathsf{let}\ (B, \Lambda) = \mathsf{eigen}(A)\ \mathsf{in}\ \ldots$$

This models how the eigen operation works in the languages R and MATLAB. We agree that Λ, like B, is the zero matrix if A is not diagonalizable.

▶ **Example 16** (Rank of a matrix)**.** Since the rank of a diagonalizable matrix equals the number of nonzero entries in its diagonal form, we can express the rank of a diagonalizable matrix A as follows:

$$\mathsf{let}\ (B, \Lambda) = \mathsf{eigen}(A)\ \mathsf{in}\ \mathbf{1}(A)^* \cdot \mathsf{apply}[\neq 0](\Lambda) \cdot \mathbf{1}(A).$$

▶ **Example 17** (Graph partitioning)**.** A well-known heuristic for partitioning an undirected graph without self-loops is based on an eigenvector corresponding to the second-smallest eigenvalue of the Laplacian matrix [27]. The Laplacian L can be derived from the adjacency matrix A as $\mathsf{let}\ D = \mathsf{diag}(A \cdot \mathbf{1}(A))\ \mathsf{in}\ \mathsf{apply}[-](D, A)$. (Here D is the degree matrix.) Now let $(B, \Lambda) \in \mathsf{eigen}(L)$. In an analogous way to Example 6, we can compute a matrix E, obtained from Λ by replacing the occurrences of the second-smallest eigenvalue by 1 and all other entries by 0. Then the eigenvectors corresponding to this eigenvalue can be isolated from B (and the other eigenvectors zeroed out) by multiplying $B \cdot E$. ◀

It turns out that MATLANG + inv is subsumed by MATLANG + eigen.

▶ **Theorem 18.** *Matrix inversion is expressible in* MATLANG + eigen*.*

A very interesting open problem is the following: *Are there graph queries expressible deterministically in* MATLANG + eigen*, but not in* MATLANG + inv*?* This is an interesting question for further research. The answer may depend on the functions that can be used in pointwise applications.

▶ Remark (Determinacy). The stipulation *deterministically* in the above open question is important. Ideally, we use the nondeterministic eigen operation only as an intermediate construct. It is an aid to achieve a powerful computation, but the final expression should have only a single possible output on every input. The expression of Example 16 is deterministic in this sense, as is the expression for inversion underlying the proof of Theorem 18.

6 The evaluation problem

The evaluation problem asks, given an input instance I and an expression e, to compute the result $e(I)$. There are some issues with this naive formulation, however. Indeed, in our theory we have been working with arbitrary complex numbers. How do we even represent the input? Notably, the eigen operation on a matrix with only rational entries may produce irrational entries. In fact, the eigenvalues of an adjacency matrix (even of a tree) need not even be definable in radicals [13]. Practical systems, of course, apply techniques from numerical mathematics to compute rational approximations. But it is still theoretically interesting to consider the exact evaluation problem.

Our approach is to represent the output symbolically, following the idea of constraint query languages [21, 25]. Specifically, we can define the input-output relation of an expression, for given dimensions of the input matrices, by an existential first-order logic formula over the reals. Such formulas are built from real variables, integer constants, addition, multiplication, equality, inequality ($<$), disjunction, conjunction, and existential quantification.

▶ **Example 19.** Consider the expression $\mathsf{eigen}(M)$ over the schema consisting of a single matrix variable M. Any instance I where $I(M)$ is an $n \times n$ matrix A can be represented by a tuple of $2 \times n \times n$ real numbers. Indeed, let $a_{i,j} = \Re A_{i,j}$ (the real part of a complex number), and let $b_{i,j} = \Im A_{i,j}$ (the imaginary part). Then $I(M)$ can be represented by the tuple $(a_{1,1}, b_{1,1}, a_{1,2}, b_{1,2}, \ldots, a_{n,n}, b_{n,n})$. Similarly, any $B \in \mathsf{eigen}(A)$ can be represented by a similar tuple. We introduce the variables $x_{M,i,j,\Re}$, $x_{M,i,j,\Im}$, $y_{i,j,\Re}$, and $y_{i,j,\Im}$, for $i, j \in \{1, \ldots, n\}$, where the x-variables describe an arbitrary input matrix and the y-variables describe an arbitrary possible output matrix. Denoting the input matrix by $[\bar{x}]$ and the output matrix by $[\bar{y}]$, we can now write an existential formula expressing that $[\bar{y}]$ is a possible result of eigen applied to $[\bar{x}]$:

- To express that $[\bar{y}]$ is a basis, we write that there exists a nonzero matrix $[\bar{z}]$ such that $[\bar{y}] \cdot [\bar{z}]$ is the identity matrix. It is straightforward to express this condition by a formula.
- To express, for each column vector v of $[\bar{y}]$, that v is an eigenvector of $[\bar{x}]$, we write that there exists λ such that $[\bar{x}] \cdot v = \lambda[\bar{x}]$.
- The final and most difficult condition to express is that distinct eigenvectors v and w that correspond to a same eigenvalue are orthogonal. We cannot write $\exists\lambda([\bar{x}] \cdot v = \lambda v \wedge [\bar{x}] \cdot w = \lambda w) \rightarrow v^* \cdot w = 0$, as this is not a proper existential formula. (Note though that the conjugate transpose of v is readily expressed.) Instead, we avoid an explicit quantifier and replace the antecedent by the conjunction, over all positions i, of $v_i \neq 0 \neq w_i \rightarrow ([\bar{x}] \cdot v)_i / v_i = ([\bar{x}] \cdot w)_i / w_i$.
- A final detail is that we should also be able to express that $[\bar{x}]$ is not diagonalizable, for in that case we need to define $[\bar{y}]$ to be the zero matrix. Nondiagonalizability is equivalent to the existence of a Jordan form with at least one 1 on the superdiagonal. We can express this as follows. We postulate the existence of an invertible matrix $[\bar{z}]$ such that the product $[\bar{z}] \cdot [\bar{x}] \cdot [\bar{z}]^{-1}$ has all entries zero, except those on the diagonal and the superdiagonal. The entries on the superdiagonal can only by 0 or 1, with at least one 1. Moreover, if an entry i, j on the superdiagonal is nonzero, the entries i, i and j, j must be equal. ◀

The approach taken in the above example leads to the following general result. The operations of MATLANG are handled using similar ideas as illustrated above for the eigen operation, and are actually easier. The let-construct, and the composition of subexpressions into larger expression, are handled by existential quantification.

▶ **Theorem 20.** *An input-sized expression consists of a schema $\mathcal{S}$, an expression e in* MATLANG + eigen *that is well-typed over $\mathcal{S}$ with output type $t_1 \times t_2$, and a size assignment σ defined on the size symbols occurring in $\mathcal{S}$. There exists a polynomial-time computable translation that maps any input-sized expression as above to an existential first-order formula ψ over the vocabulary of the reals, expanded with symbols for the functions used in pointwise applications in e, such that*

1. *Formula ψ has the following free variables:*
   - *For every $M \in \mathrm{var}(\mathcal{S})$, let $\mathcal{S}(M) = s_1 \times s_2$. Then ψ has the free variables $x_{M,i,j,\Re}$ and $x_{M,i,j,\Im}$, for $i = 1, \ldots, \sigma(s_1)$ and $j = 1, \ldots, \sigma(s_2)$.*
   - *In addition, ψ has the free variables $y_{i,j,\Re}$ and $y_{i,j,\Im}$, for $i = 1, \ldots, \sigma(t_1)$ and $j = 1, \ldots, \sigma(t_2)$.*

   The set of these free variables is denoted by $\mathrm{FV}(\mathcal{S}, e, \sigma)$.
2. *Any assignment ρ of real numbers to these variables specifies, through the x-variables, an instance I conforming to $\mathcal{S}$ by σ, and through the y-variables, a $\sigma(t_1) \times \sigma(t_2)$ matrix B.*
3. *Formula ψ is true over the reals under such an assignment ρ, if and only if $B \in e(I)$.*

The existential theory of the reals is decidable; actually, the full first-order theory of the reals is decidable [2, 4]. But, specifically the class of problems that can be reduced in polynomial time to the existential theory of the reals forms a complexity class on its own, known as $\exists\mathbf{R}$ [34, 35]. The above theorem implies that the *partial evaluation problem for* MATLANG + eigen belongs to this complexity class. We define this problem as follows. The idea is that an arbitrary specification, expressed as an existential formula χ over the reals, can be imposed on the input-output relation of an input-sized expression.

▶ **Definition 21.** The *partial evaluation problem* is a decision problem that takes as input:

- an input-sized expression $(\mathcal{S}, e, \sigma)$, where all functions used in pointwise applications are explicitly defined using existential formulas over the reals;
- an existential formula χ with free variables in $\mathrm{FV}(\mathcal{S}, e, \sigma)$ (see Theorem 20).

The problem asks if there exists an instance I conforming to $\mathcal{S}$ by σ and a matrix $B \in e(I)$ such that (I, B) satisfies χ.

For example, χ may completely specify the matrices in I by giving the values of the entries as rational numbers, and may express that the output matrix has at least one nonzero entry.

An input $(\mathcal{S}, e, \sigma, \chi)$ is a yes-instance to the partial evaluation problem precisely when the existential sentence $\exists \mathrm{FV}(\mathcal{S}, e, \sigma)(\psi \wedge \chi)$ is true in the reals, where ψ is the formula obtained by Theorem 20. Hence we can conclude:

▶ **Corollary 22.** *The partial evaluation problem for* MATLANG + eigen *belongs to $\exists\mathbf{R}$.*

Since the full theory of the reals is decidable, our theorem implies many other decidability results. We give just two examples.

▶ **Corollary 23.** *The equivalence problem for input-sized expressions is decidable. This problem takes as input two input-sized expressions $(\mathcal{S}, e_1, \sigma)$ and $(\mathcal{S}, e_2, \sigma)$ (with the same $\mathcal{S}$ and σ) and asks if for all instances I conforming to $\mathcal{S}$ by σ, we have $B \in e_1(I) \Leftrightarrow B \in e_2(I)$.*

Note that the equivalence problem for MATLANG expressions on arbitrary instances (size not fixed) is undecidable by Theorem 9, since equivalence of FO^3 formulas over binary relational vocabularies is undecidable [15].

▶ **Corollary 24.** *The determinacy problem for input-sized expressions is decidable. This problem takes as input an input-sized expression $(\mathcal{S}, e, \sigma)$ and asks if for every instance I conforming to $\mathcal{S}$ by σ, there exists at most one $B \in e(I)$.*

Corollary 22 gives an $\exists\mathbf{R}$ upper bound on the combined complexity of query evaluation [38]. Our final result is a matching lower bound, already for data complexity alone.

▶ **Theorem 25.** *There exists a fixed schema $\mathcal{S}$ and a fixed expression e in* MATLANG + eigen*, well-typed over $\mathcal{S}$, such that the following problem is hard for $\exists\mathbf{R}$: Given an integer instance I over $\mathcal{S}$, decide whether the zero matrix is a possible result of $e(I)$. The pointwise applications in e use only simple functions definable by quantifier-free formulas over the reals.*

▶ Remark (Complexity of deterministic expressions). Our proof of Theorem 25 relies on the nondeterminism of the eigen operation. Coming back to our remark on determinacy at the end of the previous section, it is an interesting question for further research to understand not only the expressive power but also the complexity of the evaluation problem for *deterministic* MATLANG + eigen expressions.

7 Conclusion

There is a commendable trend in contemporary database research to leverage, and considerably extend, techniques from database query processing and optimization, to support large-scale linear algebra computations. In principle, data scientists could then work directly in SQL or related languages. Still, some users will prefer to continue using the matrix sublanguages they are more familiar with. Supporting these languages is also important so that existing code need not be rewritten.

From the perspective of database theory, it then becomes relevant to understand the expressive power of these languages as well as possible. In this paper we have proposed a framework for viewing matrix manipulation from the point of view of expressive power of database query languages. Moreover, our results formally confirm that the basic set of matrix operations offered by systems in practice, formalized here in the language MATLANG + inv + eigen, really is adequate for expressing a range of linear algebra techniques and procedures.

In the paper we have already mentioned some intriguing questions for further research. Deep inexpressibility results have been developed for logics with rank operators [30]. Although these results are mainly concerned with finite fields, they might still provide valuable insight in our open questions. Also, we have not covered all standard constructs from linear algebra. For instance, it may be worthwhile to extend our framework with the operation of putting matrices in upper triangular form, with the Gram-Schmidt procedure (which is now partly hidden in the eigen operation), and with the singular value decomposition.

Finally, we note that various authors have proposed to go beyond matrices, introducing data models and algebra for tensors or multidimensional arrays [31, 22, 32]. When moving to more and more powerful and complicated languages, however, it becomes less clear at what point we should simply move all the way to full SQL, or extensions of SQL with recursion.

References

1 S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2 D.S. Arnon. Geometric reasoning with logic and algebra. *Artificial Intelligence*, 37:37–60, 1988.
3 S. Axler. *Linear Algebra Done Right*. Springer, third edition, 2015.
4 S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer, second edition, 2008.
5 M. Boehm, M.W. Dusenberry, D. Eriksson, A.V. Evfimievski, F.M. Manshadi, N. Pansare, B. Reinwald, F.R. Reiss, P. Sen, A.C. Surve, and S. Tatikonda. SystemML: Declarative machine learning on Spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.
6 A. Bonato. *A Course on the Web Graph*, volume 89 of *Graduate Studies in Mathematics*. American Mathematical Society, 2008.
7 S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998.
8 L. Chen, A. Kumar, J. Naughton, and J.M. Patel. Towards linear algebra over normalized data. *Proceedings of the VLDB Endowment*, 10(11):1214–1225, 2017.
9 S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in DynFO. In M.M. Halldórsson, K. Iwama, N. Kobayashi, and B. Speckmann, editors, *Proceedings 42nd International Colloquium on Automata, Languages and Programming, Part II*, volume 9135 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2015.
10 A. Dawar. On the descriptive complexity of linear algebra. In W. Hodges and R. de Queiroz, editors, *Logic, Language, Information and Computation, Proceedings 15th WoLLIC*, volume 5110 of *Lecture Notes in Computer Science*, pages 17–25. Springer, 2008.
11 A. Dawar, M. Grohe, B. Holm, and B. Laubner. Logics with rank operators. In *Proceedings 24th Annual IEEE Symposium on Logic in Computer Science*, pages 113–122, 2009.
12 G.M. Del Corso, A. Gulli, and F. Romani. Fast PageRank computation via a sparse linear system. *Internet Mathematics*, 2(3):251–273, 2005.
13 C.D. Godsil. Some graphs with characteristic polynomials which are not solvable by radicals. *Journal of Graph Theory*, 6:211–214, 1982.
14 G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, fourth edition, 2013.
15 E. Grädel, E. Rosen, and M. Otto. Undecidability results on two-variable logics. *Archive of Mathematical Logic*, 38:313–354, 1999.
16 D.J. Hand, H. Mannila, and P. Smyth. *Principles of Data Mining*. MIT Press, 2001.
17 Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001. doi:10.1145/502090.502100.
18 B. Holm. *Descriptive Complexity of Linear Algebra*. PhD thesis, University of Cambridge, 2010.
19 D. Hutchison, B. Howe, and D. Suciu. LaraDB: A minimalist kernel for linear and relational algebra computation. In F.N. Afrati and J. Sroka, editors, *Proceedings 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, pages 2:1–2:10, 2017.
20 K.E. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.
21 Paris C. Kanellakis, Gabriel M. Kuper, and Peter Z. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1):26–52, 1995. doi:10.1006/jcss.1995.1051.
22 M. Kim. *TensorDB and Tensor-Relational Model for Efficient Tensor-Relational Operations*. PhD thesis, Arizona State University, 2014.
23 A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *jacm*, 29(3):699–717, 1982.
24 Ph.G. Kolaitis. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*, chapter 2. Springer, 2007.

25 G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

26 B. Laubner. *The Structure of Graphs and New Logics for the Characterization of Polynomial Time*. PhD thesis, Humboldt-Universität zu Berlin, 2010.

27 J. Leskovec, A. Rajaraman, and J.D. Ullman. *Mining of Massive Datasets*. Cambridge University Press, second edition, 2014.

28 Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003. doi:10.1016/S0304-3975(02)00736-3.

29 H.Q. Ngo, X. Nguyen, D. Olteanu, and M. Schleich. In-database factorized learning. In J.L. Reutter and D. Srivastava, editors, *Proceedings 11th Alberto Mendelzon International Workshop on Foundations of Data Management*, volume 1912 of *CEUR Workshop Proceedings*, 2017.

30 W. Pakusa. *Linear Equation Systems and the Search for a Logical Characterisation of Polynomial Time*. PhD thesis, RWTH Aachen, 2015.

31 F. Rusu and Y. Cheng. A survey on array storage, query languages, and systems. arXiv:1302.0103, 2013.

32 T. Sato. Embedding Tarskian semantics in vector spaces. arXiv:1703.03193, 2017.

33 T. Sato. A linear algebra approach to datalog evaluation. *Theory and Practice of Logic Programming*, 17(3):244–265, 2017.

34 M. Schaefer. Complexity of some geometric and topological problems. In D. Eppstein and E.R. Gansner, editors, *Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 334–344. Springer, 2009.

35 M. Schaefer and D. Štefankovič. Fixed points, Nash equilibria, and the existential theory of the reals. *Theory of Computing Systems*, 60(2):172–193, 2017.

36 M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proceedings 2016 International Conference on Management of Data*, pages 3–18. ACM, 2016.

37 J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course in database queries. In *Proceedings 26th ACM Symposium on Principles of Database Systems*, pages 143–154. ACM Press, 2007.

38 M. Vardi. The complexity of relational query languages. In *Proceedings 14th ACM Symposium on the Theory of Computing*, pages 137–146, 1982.