MATLANG: Matrix operations and their expressive power

DOI: 10.1145/3371316.3371331
Handle: http://hdl.handle.net/1942/30024

MATLANG: Matrix operations and their expressive power

Robert Brijder
Hasselt University
Hasselt, Belgium
robert.brijder@uhasselt.be

Floris Geerts
University of Antwerp
Antwerp, Belgium
floris.geerts@uantwerpen.be

Jan Van den Bussche
Hasselt University
Hasselt, Belgium
jan.vandenbussche@uhasselt.be

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 for 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. It is defined such that for each eigenvalue a set of orthogonal eigenvectors is returned that span the eigenspace of that eigenvalue. 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. Finally, the evaluation problem for MATLANG + eigen is shown to be complete for the complexity class $\exists\mathbf{R}$.

1. INTRODUCTION

In view of the importance of large-scale statistical and machine learning (ML) algorithms in the overall data analytics workflow, database systems are in the process of being redesigned and extended to allow for a seamless integration of ML algorithms and mathematical and statistical frameworks, such as R, SAS, and MATLAB, with existing data manipulation and data querying functionality [42, 19, 5, 38, 10, 27, 21]. In particular, data scientists often use *matrices* to represent their data, as opposed to using the relational data model, and create custom data analytics algorithms using *linear algebra*, instead of writing SQL queries. Here, linear algebra algorithms are expressed in a declarative manner by composing basic linear algebra constructs such as matrix multiplication, matrix transposition, element-wise operations on the entries of matrices, solving nonsingular systems of linear equations (matrix inversion), diagonalization (eigenvalues and eigenvectors), singular value decomposition, just to name a few. The main challenges from a database system's perspective are to ensure scalability by providing physical data independence and optimizations. We refer to [39] for an overview of the different systems addressing these challenges.

In this context, the following natural questions arise: Which linear algebra constructs need to be supported to perform certain data analytical tasks? Does the additional support for certain linear algebra operations increase the overall functionality? Can a linear algebra algorithm be rewritten, in an equivalent way, to an algorithm using a smaller number of linear algebra operations? Such questions have been extensively studied for "classical" query languages (fragments and extensions of SQL) in database theory and finite model theory [1, 26]. Indeed, the questions raised all relate to the *expressive power* of query languages. In this paper we enroll in the investigation of the expressive power of *matrix query languages*.

As a starting point we focus on matrices and matrix query languages alone, leaving the study of the expressive power of languages that operate on *both* relational data and matrices for future work. Even this "matrix only" setting turns out to be quite interesting and challenging on its own.

To set the stage, we need to formally define what we mean by a matrix query language. There has been work in finite model theory and logic to understand the capability of certain logics to express linear algebra operations [13, 12, 20]. In particular, the extent to which fixpoint logics with counting and their extension with so-called rank operators can express linear algebra has been considered. The motivation for that line of work is mainly to find a logical characterization of polynomial-time computability and less so in understanding the expressive power of specific linear algebra operations.

In this paper, we take the opposite approach in which we define a basic matrix query language, referred to as MATLANG, which is built up from *basic* linear algebra operations, supported by linear algebra systems such as R and MATLAB, and then closing these operations under *composition*. All basic linear algebra operations supported in MATLANG stem from "atomic" operations supported in these popular linear algebra packages. While many other operations are supported by these packages, we feel that they are somewhat less atomic. We present examples later

on, showing that MATLANG is indeed capable of expressing common matrix manipulations. In fact, we propose MATLANG as *an analog for matrices of the relational algebra for relations*.

To study the expressive power of MATLANG, we relate it to the relational algebra with aggregates [25, 30]. In fact, it turns out that 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. We note that the connection with three-variable logics has recently been strengthened [15].

We thus see that, for example, when data analysts want to check for four-cliques in a graph, more advanced linear algebra operations than those in MATLANG need to be considered. Similarly, extracting information related to the connectivity of graphs requires extending MATLANG. We consider two such extensions in the paper: extending MATLANG with matrix inversion (inv) and extending MATLANG with an operation (eigen) to compute eigenvectors. Since no unique set of eigenvectors exists, the eigen operation is intrinsically *non-deterministic*.

We show that MATLANG + inv is strictly more expressive than MATLANG. 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 [29, 11, 35].

We show that MATLANG + eigen can express inversion by using a deterministic MATLANG + eigen expression (i.e., despite it using eigen, it always deterministically returns the inverse of a matrix, if it exists). The argument is well known from linear algebra, but our result shows that starting from the eigenvectors, MATLANG is expressive enough to construct the inverse.

We subsequently show that the *equivalence* of MATLANG + eigen expressions is decidable. Related to this is the question whether the *evaluation* of expressions in MATLANG + eigen is effectively computable. This may seem like an odd question, since linear algebra computations are done in practice. These evaluation algorithms, however, often use techniques from numerical mathematics [17], resulting in approximations of the precise result — here, we are interested in the exact result. In particular, we show that the input-output relation of an expression e in MATLANG + eigen, applied to input matrices of given dimensions, is definable in the *existential theory of the real numbers* (which is decidable [3, 4]), by a formula of size polynomial in the size of e and the given dimensions.

We finally show that, conversely, there exists a fixed expression (data complexity) in MATLANG + eigen for which the evaluation problem is $\exists\mathbf{R}$-complete, where $\exists\mathbf{R}$ is the class of problems that can be reduced in polynomial time to the existential theory of the reals [36, 37], even when restricted to input matrices with integer entries.

1.1 Related work

Programming languages to manipulate matrices trace back to the APL language [22]. Providing database support for matrices and multidimensional arrays has been a long-standing research topic [33], originally geared towards applications in scientific data management.

In [27], LARA is proposed as a domain-specific programming language written in Scala that provides both linear algebra (LA) and relational algebra (RA) constructs. This approach is taken one step further in [21] where it is shown that the RA operations and a number of LA operations can be defined in terms of three core operations called EXT, UNION, and JOIN.

Another relevant related work is the FAQ framework [2], which focuses on the project-join fragment of the algebra for K-relations [18]. The connection between MATLANG and the algebra for K-relations is more deeply investigated in [8]. Yet another related formalism is that of logics with rank operators [13, 12, 32]. These operators solve 0, 1-matrices over finite fields, and increase the expressive power of established logics over abstract structures. In contrast, in this paper we are interested in queries on arbitrary matrices.

Modest changes to SQL in order to perform LA operations in a scalable way within relational databases are proposed in [31]. In this way, various linear algebra operations are implemented in an efficient way using the relational algebra.

While the previous work is focused on showing that relational algebra (appropriately extended) can serve as a platform for supporting large scale linear algebra operations, the focus of our work here is complementary. Indeed, we want to understand the precise expressive power of common linear algebra operations, as adequately formalized in the language MATLANG and its extensions (see [7] for more details).

2. MATLANG

2.1 Syntax and semantics

We assume a sufficient supply of *matrix variables*, which serve to indicate the inputs to expressions in MATLANG. 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. We will see various examples of MATLANG expressions below.

To define the semantics of MATLANG, we first define the basic matrix operations. Following practical matrix sublanguages such as those of 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.

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.

$$\mathbf{1}\begin{pmatrix}2 & \sqrt{3} & 4\\ 4 & 5 & 6\end{pmatrix} = \begin{pmatrix}1\\1\end{pmatrix} \qquad \mathsf{diag}\begin{pmatrix}6\\7\end{pmatrix} = \begin{pmatrix}6 & 0\\0 & 7\end{pmatrix}$$

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

Figure 1: Some basic matrix operations of MATLANG.

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 2.1. *The operations $\mathbf{1}(A)$, $\mathsf{diag}(v)$, and $\mathsf{apply}[f](A^{(1)}, \ldots, A^{(n)})$ 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.*

The formal semantics of MATLANG expressions is defined in a straightforward manner. Expressions will be evaluated over instances where 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 $\mathrm{var}(I)$. The rules that allow to derive that an expression e, on an instance I, *successfully evaluates* to a matrix A, denoted by $e(I) = A$, is defined recursively as follows. If $M \in \mathrm{var}(I)$, then $M(I) := I(M)$. If $e_1(I) = A$ and $e_2(I[M := A]) = B$, where $I[M := A]$ is the instance obtained from I by mapping M to the matrix A, then $(\mathsf{let}\ M = e_1\ \mathsf{in}\ e_2)(I) := B$. We have $e^*(I) := (e(I))^*$, $(\mathbf{1}(e))(I) := \mathbf{1}(e(I))$, and if $e(I)$ is a column vector, then $(\mathsf{diag}(e))(I) := \mathsf{diag}(e(I))$. Moreover, if the number of columns of $e_1(I)$ is equal to the number of rows of $e_2(I)$, then $(e_1 \cdot e_2)(I) := e_1(I) \cdot e_2(I)$. Finally, if $e_k(I)$ for $k \in \{1, \ldots, n\}$ all have the same dimensions, then $\mathsf{apply}[f](e_1, \ldots, e_n) := \mathsf{apply}[f](e_1(I), \ldots, e_n(I))$.

The reason why an evaluation may not succeed (i.e., $e(I)$ may not be defined) is that 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 2.2 (SCALARS). *As a first example we show how to express scalars (elements in $\mathbf{C}$). Obviously, in practice, scalars would be part of the language. In this paper, however, we are interested in expressiveness, so we start from a minimal language (MATLANG) and then see what is expressible in this language. To express a scalar $c \in \mathbf{C}$, consider (by abuse of notation) the constant function $c : \mathbf{C} \to \mathbf{C} : z \mapsto c$ and the MATLANG expression*

$$c := \mathsf{apply}[c]\big(\mathbf{1}(\mathbf{1}(M)^*)\big).$$

Regardless of the matrix assigned to M, the expression evaluates to the 1×1 matrix whose unique entry is scalar c.

EXAMPLE 2.3 (SCALAR MULTIPLICATION). *We can also express scalar multiplication of a matrix by a scalar, i.e., the operation which multiplies every entry of a matrix by the same scalar. Indeed, let c be a scalar and consider the MATLANG expression*

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

where c is the scalar expression from the previous example. If M is assigned an $m \times n$ matrix A, then $c(A)$ returns the 1×1 matrix $[c]$ and in variable O we compute the $m \times n$ matrix where every entry equals c. Then pointwise multiplication $\times$ with returns $x \times y$ on input (x, y) is used to do the scalar multiplication of A by c. This example generalizes in a straightforward manner to

$$\mathsf{apply}[\times]\big(\mathbf{1}(e_2) \cdot e_1 \cdot (\mathbf{1}(e_2^*))^*, e_2\big),$$

where e_1 and e_2 are MATLANG expressions such that $e_1(I)$ is a 1×1-matrix for any instance I. It should be clear that this expression evaluates to the scalar multiplication of $e_2(I)$ by $e_1(I)$ for any I. We use $e_1 \odot e_2$ as a shorthand notation for this expression. For example, $c \odot e_2$ represents the scalar multiplication of e_2 by the scalar c.

EXAMPLE 2.4 (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 [9, 6] of A is the $n \times n$ matrix G defined by $G_{i,j} = dA_{ij}/k_i + (1-d)/n$. The calculation of G from A can be expressed in MATLANG as follows:*

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

In variable J we compute the $n \times n$ matrix where every entry equals one. In $A \cdot J$ we compute the $n \times n$ matrix where all entries in the ith row equal k_i. An $n \times n$ matrix holding the entries A_{ij}/k_i is computed in B. 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 2.2, and $\odot$ from Example 2.3.

2.2 Types and schemas

We now introduce a notion of schema, which assigns types to matrix names, so that expressions can be type-checked against schemas. We already remarked the need for this. Indeed, 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.

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 type-checked. Our types also need to be able to recognize vectors, so that diag can be type-checked.

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 rules that allow to derive that an expression e over a schema $\mathcal{S}$ *successfully infers* an output type τ, denoted by $\mathcal{S} \vdash e : \tau$, are defined recursively as follows. If $M \in \mathrm{var}(\mathcal{S})$, then $\mathcal{S} \vdash M : \mathcal{S}(M)$. If $\mathcal{S} \vdash e_1 : \tau_1$ and $\mathcal{S}[M := \tau_1] \vdash e_2 : \tau_2$, where $\mathcal{S}[M := \tau]$ denotes the schema that is obtained from $\mathcal{S}$ by mapping M to the type τ, then $\mathcal{S} \vdash \mathsf{let}\ M = e_1\ \mathsf{in}\ e_2 : \tau_2$. If $\mathcal{S} \vdash e : s_1 \times s_2$, then $\mathcal{S} \vdash e^* : s_2 \times s_1$ and $\mathcal{S} \vdash \mathbf{1}(e) : s_1 \times 1$. If $\mathcal{S} \vdash e : s \times 1$, then $\mathcal{S} \vdash \mathsf{diag}(e) : s \times s$. If $\mathcal{S} \vdash e_1 : s_1 \times s_2$ and $\mathcal{S} \vdash e_2 : s_2 \times s_3$, then $\mathcal{S} \vdash e_1 \cdot e_2 : s_1 \times s_3$. Finally, $\mathcal{S} \vdash e_k : \tau$ for $k \in 1, \ldots, n$ with $n > 0$ and $f : \mathbf{C}^n \to \mathbf{C}$, then $\mathcal{S} \vdash \mathsf{apply}[f](e_1, \ldots, e_n) : \tau$.

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.

PROPOSITION 2.5 (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

3.1 Relational representation of matrices

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 [30]. 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. We now rephrase our relational encoding more formally in this setting.

That is, 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 $\mathrm{var}(\mathcal{S})$.

To define relational instances, we assume a countably infinite universe **dom** of abstract atomic data elements. For notational convenience, we assume that the natural numbers are contained in **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 $R \in \mathrm{var}(\mathcal{S})$.

The matrix data model can now be formally connected to the relational data model, as follows. 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

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

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

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

3.2 To relational algebra with summation

Given the representation of matrices by relations, we now show that MATLANG can be simulated in the relational algebra with aggregates. Actually, the only aggregate operation we need is summation. The relational algebra with summation extends the well-known relational algebra for relational databases and is defined as follows. For a full formal definition, see [30]. For our purposes it suffices to highlight the following about the relational algebra with summation. 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. Function application and summation are defined as follows.

- 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 $\{i_1, \ldots, i_n\}$ as a subset of its set of numerical columns. 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 $n = 0$, i.e., constants f.

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

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

where $\mathsf{group}[i_1, \ldots, i_n](r, t)$ is equal to

$$\big\{t' \in r \mid t'(i_1) = t(i_1) \wedge \cdots \wedge t'(i_n) = t(i_n)\big\}.$$

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

Given that relations are typed, 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 [40] and is analogous to what we have already done for MATLANG in Section 2.2.

THEOREM 3.1. *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.

3.3 To relational calculus with summation

We can sharpen Theorem 3.1 by working in the *relational calculus* with aggregates. In this logic, we have base variables and numerical variables. Base variables can be bound to base columns of relations, and compared for equality. Numerical variables can be bound to numerical columns, and can be equated to function applications and aggregates. We will not recall the syntax formally, see [30] for a full definition. It turns out that when simulating MATLANG expression in the relational calculus with aggregates we only need formulas with at most *three* base variables.

PROPOSITION 3.2. *Let $\mathcal{S}$, e, τ, and ℓ as in Theorem 3.1. For every* MATLANG *expression e there is a formula φ_e over $Rel(\mathcal{S})$ in the relational calculus with summation, such that*

1. *If τ is general, $\varphi_e(i, j, z)$ has two free base variables i and j and one free numerical variable z; if τ is a vector type, we have $\varphi_e(i, z)$; and if τ is scalar, we have $\varphi_e(z)$.*
2. *For every instance I, the relation defined by φ_e on $Rel(I)$ equals $Rel_\ell(e(I))$.*
3. *The formula φ_e uses only three distinct base variables. The functions used in pointwise applications in φ_e are as in the statement of Theorem 3.1. Furthermore, φ_e neither uses equality conditions between numerical variables nor equality conditions on base variables involving constants.*

3.4 Expressing graph queries

We now express relational queries as matrix queries. This works best for binary relations, or graphs, which we can represent by their adjacency matrices.

Formally, we 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 (i.e., the domain elements that occur in some tuple of some relation of I) equals $\{1, \ldots, n\}$ for some positive natural number n.

To every graph schema $\mathcal{S}$ we associate a matrix schema $Mat(\mathcal{S})$, where $(Mat(\mathcal{S}))(R) = \alpha \times \alpha$ for every $R \in \mathrm{var}(\mathcal{S})$, 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 $R \in \mathrm{var}(\mathcal{S})$.

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 3.1. We assume active-domain semantics for first-order logic [1]. Note that the following result deals only with pure first-order logic, without aggregates or numerical columns.

THEOREM 3.3. *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 3.4. *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.

We conclude by showing that MATLANG cannot express the transitive-closure graph query which maps a graph to its transitive closure. This follows from the locality of the calculus with aggregates [30].

PROPOSITION 3.5. *The graph query over a single binary relation R that maps I to the transitive-closure of $I(R)$ is not expressible in* MATLANG.

4. MATRIX INVERSION

We now consider the extension of MATLANG with matrix inversion. 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 $\mathsf{MATLANG} + \mathsf{inv}$.

EXAMPLE 4.1 (PAGERANK). *Recall Example 2.4 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. We use e_B and e_n to denote the corresponding* MATLANG *expressions. 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 [14]:*

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

This calculation is readily expressed in MATLANG + inv *as*

$$(1-d) \odot (\mathsf{apply}[1/x](e_n)) \odot \big(\mathsf{apply}[-](\mathsf{diag}(\mathbf{1}(M)), d \odot e_B)\big)^{-1} \cdot \mathbf{1}(M).$$

EXAMPLE 4.2 (TRANSITIVE CLOSURE). *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 $\mathsf{diag}(\mathbf{1}(A))$. Let e_n be the expression computing the scalar n. The sum of the absolute values of the entries of each column of $B = \frac{1}{n+1}A$ is strictly less than 1, so $S = \sum_{k=0}^{\infty} B^k$ converges, and is equal to $(I-B)^{-1}$ [17, 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 compute the reflexive-transitive closure of r by evaluating*

$$\mathsf{let}\ M = \mathsf{apply}[-]\big(\mathsf{diag}(\mathbf{1}(M)), \mathsf{apply}[1/(x+1)](e_n) \odot M\big)\ \mathsf{in}\ \mathsf{apply}[\neq 0](M^{-1})$$

by assigning matrix variable M to A. Here, $\neq 0$ is the function which returns 1 if the value is nonzero and 0 otherwise. We can express the transitive closure by multiplying the above expression by M.

Given our earlier observation that the transitive-closure query cannot be expressed in MATLANG (Proposition 3.5) and the MATLANG + inv expression given in the previous example which does express this query, we may conclude:

THEOREM 4.3. 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, and checking cyclicity. Using Theorem 3.3 one can show that MATLANG is able to reduce these queries to the transitive-closure query.

5. EIGENVECTORS

We next consider the extension of MATLANG with an operation eigen. 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 the 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. In fact, there are infinitely many eigenvectors.

Hence, some care must be taken in extending MATLANG with the eigen operation. 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 semantic rules of MATLANG 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.

EXAMPLE 5.1 (RANK OF A MATRIX). *First, we remark that one can show that the diagonal matrix containing the eigenvalues Λ corresponding to the matrix B of eigenvectors computed by $\mathsf{eigen}(A)$ is expressible in* MATLANG + eigen. *Hence we allow a shorthand notation where $\mathsf{eigen}(A)$ obtains the tuple (B, Λ) instead of just B. We then agree that Λ, like B, is a zero matrix if A is not diagonalizable.*

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:

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

Using a known argument from linear algebra we obtain that MATLANG + inv is subsumed by MATLANG + eigen.

THEOREM 5.2. *Matrix inversion is expressible in* MATLANG + eigen.

An interesting open problem is the following: *Are there graph queries expressible deterministically in* MATLANG + eigen, *but not in* MATLANG + inv*?*

6. THE EVALUATION PROBLEM

We next consider the evaluation problem of expressions in our most expressive language MATLANG + eigen. Naively, 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 [16]. 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. For a treatise on computations of eigenvectors, inverses, and other matrix notions, we refer to [17].

Our approach is to represent the output symbolically, following the idea of constraint query languages [23, 28]. 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.

Any $m \times n$ matrix A can be represented by a tuple of $2mn$ 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 A can be represented by the tuple $(a_{1,1}, b_{1,1}, a_{1,2}, b_{1,2}, \ldots, a_{m,n}, b_{m,n})$. The next result introduces the variables $x_{M,i,j,\Re}$, $x_{M,i,j,\Im}$, $y_{i,j,\Re}$, and $y_{i,j,\Im}$, where the x-variables describe an arbitrary input matrix $I(M)$ and the y-variables describe an arbitrary possible output matrix $e(I)$.

In the following, an *input-sized expression* consists of a schema $\mathcal{S}$, an expression e in $\mathsf{MATLANG} + \mathsf{eigen}$ 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 $\mathcal{S}$. For complexity considerations, we assume the sizes given in σ are coded in unary.

THEOREM 6.1. *There exists a polynomial-time computable translation that maps any input-sized expression e to an existential first-order formula ψ_e over the vocabulary of the reals, expanded with symbols for the functions used in pointwise applications in e, such that*

1. *Formula ψ_e has the following free variables:*
   - *For every $M \in \mathrm{var}(\mathcal{S})$, let $\mathcal{S}(M) = s_1 \times s_2$. Then ψ_e 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_e has the free variables $y_{e,i,j,\Re}$ and $y_{e,i,j,\Im}$, 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_e 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 [3, 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 $\exists\mathbf{R}$ [36, 37]. This class lies between NP and PSPACE. The above theorem implies that the *intensional evaluation problem for* $\mathsf{MATLANG} + \mathsf{eigen}$ 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 6.2. *The* intensional evaluation problem *is a decision problem that takes as input: (1) an input-sized expression $(\mathcal{S}, e, \sigma)$, where all functions used in pointwise applications are explicitly defined using existential formulas over the reals, and (2) an existential formula χ with free variables in* $\mathrm{FV}(\mathcal{S}, e, \sigma)$.

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.

An input $(\mathcal{S}, e, \sigma, \chi)$ is a yes-instance to the intensional evaluation problem precisely when the existential sentence $\exists \mathrm{FV}(\mathcal{S}, e, \sigma)(\psi_e \wedge \chi)$ is true in the reals, where ψ_e is the formula obtained by Theorem 6.1. Hence we can conclude:

COROLLARY 6.3. *The intensional evaluation problem for* $\mathsf{MATLANG} + \mathsf{eigen}$ *belongs to* $\exists\mathbf{R}$.

Since the full first-order theory of the reals is decidable, our theorem implies many other decidability results, including that both the equivalence problem and the determinacy problem for input-sized expressions are decidable.

Corollary 6.3 gives an $\exists\mathbf{R}$ upper bound on the combined complexity of query evaluation [41]. Our final result is a matching lower bound, already for data complexity alone.

THEOREM 6.4. *There exists a fixed schema $\mathcal{S}$ and a fixed expression e in* $\mathsf{MATLANG} + \mathsf{eigen}$*, 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.*

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 languages 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. Our results formally confirm that the basic set of matrix operations offered by systems in practice, formalized here in the language $\mathsf{MATLANG} + \mathsf{inv} + \mathsf{eigen}$, really is adequate for expressing a range of linear algebra techniques and procedures.

Deep inexpressibility results have been developed for logics with rank operators [32]. 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.

There also have been proposals to go beyond matrices, introducing data models and algebra for tensors or multi-dimensional arrays [33, 24, 34]. It would be interesting to understand the expressive power of such tensor languages.

Acknowledgments.

We thank Bart Kuijpers, Lauri Hella, Wied Pakusa, Christoph Berkholz, and Anuj Dawar for helpful discussions, and Wim Martens for useful comments on the text. R.B. is a postdoctoral fellow of the Research Foundation – Flanders (FWO).

8. REFERENCES

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

[2] M. Abo Khamis, H. Ngo, and A. Rudra. FAQ: questions asked frequently. In *Proc. PODS 2016*. ACM Press, 2016.

[3] D. Arnon. Geometric reasoning with logic and algebra. *Artif. Intell.*, 37:37–60, 1988.

[4] S. Basu, R. Pollack, and M.-F. Roy. *Algorithms in Real Algebraic Geometry*. Springer, second edition, 2008.

[5] M. Boehm et al. SystemML: declarative machine learning on Spark. *Proc. VLDB Endow*, 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] R. Brijder, F. Geerts, J. Van den Bussche, and T. Weerwag. On the expressive power of query languages for matrices. *ACM TODS*, 2019. To appear.

[8] R. Brijder, M. Gyssens, and J. Van den Bussche. On matrices and K-relations. arXiv:1904.03934, 2019.

[9] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Comput. Networks ISDN*, 30:107–117, 1998.

[10] L. Chen, A. Kumar, J. Naughton, and J. Patel. Towards linear algebra over normalized data. *Proc. VLDB Endow*, 10(11):1214–1225, 2017.

[11] S. Datta, R. Kulkarni, A. Mukherjee, T. Schwentick, and T. Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018.

[12] A. Dawar. On the descriptive complexity of linear algebra. In W. Hodges and R. de Queiroz, editors, *Proc. WoLLIC 2008*, volume 5110 of *LNCS*, pages 17–25. Springer, 2008.

[13] A. Dawar, M. Grohe, B. Holm, and B. Laubner. Logics with rank operators. In *Proc. LICS 2009*, pages 113–122, 2009.

[14] G. Del Corso, A. Gulli, and F. Romani. Fast PageRank computation via a sparse linear system. *Internet Math.*, 2(3):251–273, 2005.

[15] F. Geerts. On the expressive power of linear algebra on graphs. In *Proc. ICDT 2019*, volume 127 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019.

[16] C. Godsil. Some graphs with characteristic polynomials which are not solvable by radicals. *J. Graph Theory*, 6:211–214, 1982.

[17] G. Golub and C. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, fourth edition, 2013.

[18] T. Green, G. Karvounarakis, and V. Tannen. Provenance semirings. In *Proc. PODS 2007*, pages 31–40. ACM Press, 2007.

[19] J. Hellerstein et al. The MADlib analytics library: Or MAD skills, the SQL. *Proc. VLDB Endow*, 5(12):1700–1711, 2012.

[20] B. Holm. *Descriptive Complexity of Linear Algebra*. PhD thesis, University of Cambridge, 2010.

[21] D. Hutchison, B. Howe, and D. Suciu. LaraDB: a minimalist kernel for linear and relational algebra computation. In *Proc. BeyondMR 2007*, pages 2:1–2:10, 2007.

[22] K. Iverson. *A Programming Language*. John Wiley & Sons, Inc., 1962.

[23] P. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. *J. Comput. Syst. Sci.*, 51(1):26–52, Aug. 1995.

[24] M. Kim. *TensorDB and Tensor-Relational Model for Efficient Tensor-Relational Operations*. PhD thesis, Arizona State University, 2014.

[25] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

[26] P. Kolaitis. On the expressive power of logics on finite models. In *Finite Model Theory and Its Applications*, chapter 2. Springer, 2007.

[27] A. Kunft, A. Alexandrov, A. Katsifodimos, and V. Markl. Bridging the gap: Towards optimization across linear and relational algebra. In *Proc. BeyondMR 2016*, pages 1:1–1:4, 2016.

[28] G. Kuper, L. Libkin, and J. Paredaens, editors. *Constraint Databases*. Springer, 2000.

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

[30] L. Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296:379–404, 2003.

[31] S. Luo, Z. Gao, M. Gubanov, L. L. Perez, and C. Jermaine. Scalable linear algebra on a relational database system. In *Proc. ICDE 2017*, pages 523–534. IEEE Computer Society, 2017.

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

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

[34] T. Sato. Embedding Tarskian semantics in vector spaces. arXiv:1703.03193, 2017.

[35] T. Sato. A linear algebra approach to datalog evaluation. *Theory Pract. Log. Prog.*, 17(3):244–265, 2017.

[36] M. Schaefer. Complexity of some geometric and topological problems. In D. Eppstein and E. Gansner, editors, *Graph Drawing*, volume 5849 of *LNCS*, pages 334–344. Springer, 2009.

[37] M. Schaefer and D. Štefankovič. Fixed points, Nash equilibria, and the existential theory of the reals. *Theory Comput. Syst.*, 60(2):172–193, 2017.

[38] M. Schleich, D. Olteanu, and R. Ciucanu. Learning linear regression models over factorized joins. In *Proc. SIGMOD 2016*, pages 3–18. ACM, 2016.

[39] A. Thomas and A. Kumar. A comparative evaluation of systems for scalable linear algebra-based analytics. *Proc. VLDB Endow*, 11(13):2168–2182, 2018.

[40] J. Van den Bussche, D. Van Gucht, and S. Vansummeren. A crash course in database queries. In *Proc. PODS 2007*, pages 143–154. ACM Press, 2007.

[41] M. Vardi. The complexity of relational query languages. In *Proc. STOC 1982*, pages 137–146, 1982.

[42] Y. Zhang, W. Zhang, and J. Yang. I/O-efficient statistical computing with RIOT. In *Proc. ICDE 2010*, pages 1157–1160, 2010.