Matrix Query Languages

Floris Geerts
University of Antwerp
floris.geerts@uantwerp.be

Thomas Muñoz
PUC Chile and IMFD Chile
tfmunoz@uc.cl

Cristian Riveros
PUC Chile and IMFD Chile
cristian.riveros@uc.cl

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

Domagoj Vrgoč
PUC Chile and IMFD Chile
dvrgoc@ing.puc.cl

ABSTRACT

Due to the importance of linear algebra and matrix operations in data analytics, there has been a renewed interest in developing query languages that combine both standard relational operations and linear algebra operations. We survey aspects of the matrix query language MATLANG and extensions thereof, and connect matrix query languages to classical query languages and arithmetic circuits.

1. INTRODUCTION

Quisani:[1] You wanted to talk to me?

Authors: Indeed, Dan Olteanu asked us to write a database theory principles column for SIGMOD Record and we wonder whether you would like to help us?

Q: Sure, what is the topic?

A: Matrix query languages.

Q: I know matrices and query languages, but what do these two have to do with each other?

A: So you are familiar with the relational algebra (with aggregation), which provides the basis of SQL in the form of a canonical set of operations for manipulating relations. It is natural to wonder if there is something similar for matrices.

Q: Then I suppose matrix multiplication would definitely be a part of what you're looking for.

A: Certainly, but what else? That's what we have investigated.

Q: I'm sure you want to tell me all about it, but, sorry for asking, why do you want to do that? Can't you just view a matrix as a relation and then use relational query languages to perform matrix operations? I once wrote a SQL query for matrix multiplication as a homework. If I represent matrices $\mathbf{A}$ and $\mathbf{B}$ as ternary relations $A(i,j,\text{val})$ and $B(j,k,\text{val})$, then $\mathbf{A}\cdot\mathbf{B}$ can be computed by:

```
SELECT A.i, B.k, SUM(A.val * B.val)
FROM A, B
WHERE A.j = B.j
GROUP BY A.i, B.k;
```

So, it is not too difficult to do linear algebra in SQL?

A: Fair point. The compilation of linear algebra computations into SQL is indeed a possible approach and, in fact, underlies many older and more recent approaches to integrate matrix and relational operations in database management systems [12, 13, 33, 37, 46].

Q: I admit, though, that this does not really answer your original question. I don't suppose any arbitrary SQL query can be considered to be a matrix query.

A: In the end what is or what isn't a matrix query is probably a philosophical question. But full SQL is probably not the first what comes to the mind of a data scientist when asked what they would consider their core set of matrix operations.

Q: It may not even be a purely philosophical question. Delineating a natural core of matrix queries could inform SQL processors and help improve their performance for data science applications.

A: Now you're talking!

Q: But actually, do you have a concrete example of an SQL query that would not immediately qualify as a natural matrix query?

A: Consider counting the four-cliques in a social network. Representing the set of edges of the network as a binary relation, we can express this in SQL by a four-way join. You can, however, also represent the network by its adjacency matrix. Nevertheless, counting four-cliques would not be a typical task that you would try to solve using the matrix library of your standard data science toolbox.

Q: Hmm. I suppose though that this depends on the operations that this toolbox makes available.

[1] The conversational style used in this paper originates from Yuri Gurevich's column on Logic in Computer Science in the EATCS Bulletin. Yuri gracefully allows us to borrow his inquisitive and critical virtual student "Quisani" to guide the conversation.

A: Of course! But that is precisely what we want to understand: what can we compute, given a certain set of linear algebra operations?

Q: But wait. In textbooks on linear algebra algorithms I have seen many algorithms that rely on iterating over the indices of matrices. In a numerical package like R, we can program with for-loops. Then couldn't we count 4-cliques by simply iterating over all possible four distinct vertices and increment a counter only if such four vertices are all pairwise connected? This last check can be done using matrix multiplications of the adjacency matrix $\mathbf{A}_G$ of the network G. The for-loop program:

```
$c{=}0$
for pairwise distinct $i,j,k,\ell$ do
    $c{+}{=}\mathbf{A}_G(i,j)\cdot\mathbf{A}_G(i,k)\cdot\mathbf{A}_G(i,\ell)\cdot\mathbf{A}_G(j,k)\cdot$
        $\mathbf{A}_G(j,\ell)\cdot\mathbf{A}_G(k,\ell)$
return $c/(4!)$
```

would do the trick, where the factor $1/(4!)$ is in place because every 4-clique is counted $4!{=}24$ times.

A: Yuri told me you're sharp! Indeed, we can consider matrix query languages with iterations and we actually did so recently. But also here, we want to understand what we can compute when iterations are allowed, and what cannot be computed. So, whenever we add new constructs to our matrix query language, we want to understand the impact on the expressive power.

Q: I see. You got me hooked. Perhaps you can begin by telling how everything started?

2. MATLANG

2.1 The MATLANG Language

A: Sure. We started by inspecting common linear algebra operations supported in mathematics packages like Maple, Matlab, Mathematica, and so on. From their documentation, we extracted five operations which we believed are "atomic": matrix transposition, matrix multiplication, the computation of a vector consisting entirely out of the value 1, turning a vector into a diagonal matrix by placing it on the diagonal, and applying n-ary pointwise functions. Then, using matrix variables, as place holders for concrete matrices, and by closing our operations under composition, we obtained our initial matrix query language MATLANG [15].

Q: Are the matrices over the real or over the complex numbers?

A: We focus here on real numbers, but complex numbers can be used as well, please see our paper on that. Actually, to gauge the expressive power of matrix languages, we often just consider adjacency matrices of graphs as inputs, like we already did earlier. And in fact, since square matrices correspond to edge-weighted graphs, matrix query languages can be naturally viewed as graph query languages as well.

Q: OK. So show me Hello World in MATLANG!

A: You could call constant functions that return the ascii code of the subsequent letters :-). But instead, let us count the number of cycles of length three in a graph.

Q: Doesn't this correspond to computing the trace of $(\mathbf{A}_G)^3$, that is, summing up all the diagonal entries of the adjacency matrix $\mathbf{A}_G$ to the power of three?

A: Indeed. And we can compute this in MATLANG. Let X be a matrix variable. Then the MATLANG expression $e_1(X){:=}X\cdot X\cdot X$ using matrix multiplication, when evaluated on $\mathbf{A}_G$, will return $(\mathbf{A}_G)^3$.

The computation of the trace, which is not a basic operation in MATLANG, is now done by first constructing the diagonal identity matrix $\mathbf{I}$ of the same dimension as $\mathbf{A}_G$, then using pointwise multiplication, followed by summing up all the entries. More precisely, $e_2(X){:=}\mathsf{diag}(\mathbb{1}(X))$ will return $\mathbf{I}$ when X is assigned $\mathbf{A}_G$ because $\mathbb{1}(\mathbf{A}_G)$ returns the column vector $\mathbf{1}$ of the dimension of $\mathbf{A}_G$ consisting of all ones, by definition, and $\mathsf{diag}(\cdot)$ turns $\mathbf{1}$ into a diagonal matrix with $\mathbf{1}$ on its diagonal, i.e., it evaluates to $\mathbf{I}$.

We can now use pointwise multiplication $f_{\odot}{:}\mathbb{R}^2{\to}\mathbb{R}{:}$ $(x,y){\mapsto}x{\cdot}y$, and apply it on both $e_1(\mathbf{A}_G){=}(\mathbf{A}_G)^3$ and $e_2(\mathbf{A}_G){=}\mathbf{I}$, resulting in a diagonal matrix containing at position (i,i) the number of cycles of length three in vertex i. When we consider the final expression

$$e(X){:=}\mathbb{1}^t(X)\cdot f_{\odot}\big(X\cdot X\cdot X,\mathsf{diag}(\mathbb{1}(X))\big)\cdot\mathbb{1}(X),$$

where $\mathbb{1}^t(\mathbf{A}_G)$ returns the transpose $\mathbf{1}^t$ of $\mathbf{1}$. Then, $e(\mathbf{A}_G)$ will be the total number of cycles of length three. This is because by multiplying by $\mathbf{1}^t$ in front, and by $\mathbf{1}$ at the end, we simply sum up all entries.

Q: Cool! MATLANG does not appear to be very user-friendly, however.

A: Agreed, but it primarily serves as a formalisation to study the expressive power of linear algebra operations. We could have added, say the trace operation as a basic construct, and then we simply needed to write $e(X){:=}$ $\mathsf{tr}(X\cdot X\cdot X)$ to count cycles of length three. The point is that the trace operation is already definable using our basic operations. As such, it does not add expressive power. For the same reason, we did not include scalar multiplication, or matrix addition as basic operations.

Q: Now let me try my first MATLANG program. I believe that $e(X){:=}f_{1/6\times}(\mathsf{tr}(X^3))$, where $f_{1/6\times}{:}\mathbb{R}{\to}\mathbb{R}{:}$ $x{\mapsto}\frac{x}{6}$, counts the number of 3-cliques in a graph since every 3-clique corresponds to six cycles of length three (with different orientation and start vertex). Do you have a similar expression for 4-cliques? I do not see immediately how to do this.

2.2 Limitations of MATLANG

A: Although 3-cliques can be checked in MATLANG, as you showed, the 4-clique query is not expressible.

Q: Can you prove it?

A: For that it is instructive to think in terms of first-order logic extended with aggregates and external functions, which can serve as a formalisation of SQL [30]. To express 4-clique in that logic, we need four variables. In contrast, one can show that any MATLANG expression can be translated into this logic using three variables only.

We can now further reduce this to a logical expression, also using three variables, that only uses the edge relation, but in which infinitary disjunctions and conjunctions are allowed (see e.g., Chapter 8.6 of Libkin's book [36], or our MATLANG paper [15]). In other words, in this formula, all aggregates, function applications and basically all arithmetic operations are eliminated.

Q: So, the next step is to show that checking 4-cliques cannot be done in the three-variable fragment of infinitary first-order logic?

A: This is classic finite model theory. We can leverage the Cai-Fürer-Immerman construction [17] to obtain two graphs G and H, that agree on all sentences in that logic, yet G contains a special kind of 4-cliques, whereas H does not. The precise construction is given by Martin Otto (Example 2.7, Lemma 2.8 in [38]). This shows that a sentence for 4-cliques cannot exist.

Q: Right. So you are able to get some insights on matrix query languages based on results from classical logics. Do you also have an example of a real linear algebra operation which you cannot do in MATLANG?

A: Yes, a nice one is computing the inverse of a matrix. If you could do that, you could actually compute the transitive closure of a binary relation. But you can't do that in MATLANG.

Q: That makes sense, since MATLANG is subsumed by SQL and transitive closure is not expressible by a single select statement in non-recursive SQL [36]. But how do you reduce transitive closure to matrix inversion?

A: We use the following property: If the largest eigenvalue of a matrix $\mathbf{A}$ is strictly smaller than 1, then $(\mathbf{I}-\mathbf{A})^{-1}=\sum_{n\geq 0}\mathbf{A}^n$ holds [26]. So when $\mathbf{A}$ is an adjacency matrix $\mathbf{A}_G$, $(\mathbf{I}-\mathbf{A}_G)^{-1}$ contains non-zero values only in entries that belong to the transitive closure of G.

Q: But this only applies to matrices satisfying the eigenvalue condition?

A: Yes, but for a matrix $\mathbf{A}$ we can obtain in MATLANG its scaled version $\frac{1}{\|\mathbf{A}\|_1+1}\mathbf{A}$ where $\|\mathbf{A}\|_1$ is the sum of the absolute values of all its entries. Each eigenvalue of this matrix is strictly smaller than one [26].

Q: So since MATLANG can not specify the unbounded sum, the expression for transitive closure first computes this scaled version, subtracts it from the identity matrix, which you showed earlier how to compute in MATLANG, followed by your assumed expression for matrix inversion? You could conclude by applying some function that maps non-zero entries to 1 in order to get the true adjacency matrix of the transitive closure of G?

A: You're becoming a MATLANG expert!

2.3 MATLANG + Other Operations

Q: I am getting the hang of it! How about adding matrix inversion as an operation to MATLANG? In fact, it has been bugging me all along that the "atomic" operations you included, did not incorporate important operations related to eigenvalues and eigenvectors, matrix decompositions like (P)LU, SVD, and so on.

A: Sure, we would like to gain a better understanding of MATLANG extended with other, more complex, linear algebra operations. Unfortunately, we can't even say much yet what could be done if matrix inversion is added to MATLANG, apart from being able to compute transitive closure. Furthermore, for operations involving eigenvalues or eigenvectors, it is even challenging to define these operations in a good way.

Q: Why is that? Eigenvalues and eigenvectors are being used in practice all the time!

A: First, in which order should we return the eigenvalues? Even when starting from a real matrix, complex eigenvalues may occur. No ordering on complex numbers exists. Second, there is no unique choice for eigenvectors corresponding to an eigenvalue. Indeed, any set of vectors spanning the eigenspace could be returned. It is again unclear how to define all this in an elegant way.

Q: You could allow non-determinism by returning any set of eigenvectors that span the eigenspace.

A: That is how we defined an eigen operation and added it to MATLANG. We then focused on expressions that return a deterministic result, despite that they may use the non-deterministic eigen operation. So basically, we consider linear algebra computations that are independent of the choices of eigenvectors or orderings on eigenvalues. We can show, for example, that matrix inversion can be obtained in such a deterministic way, when MATLANG is extended with the eigen operation [15].

Q: I see, so adding eigen subsumes adding inversion. Is adding eigen strictly more powerful than adding inversion?

A: In a trivial sense, yes, because eigenvalues can be complex imaginary numbers, even when the input matrix has only zeros and ones. In contrast, the inverse of a real matrix still has real entries.

Q: It would be much more interesting to show that certain decision problems about graphs (adjacency matrices) can be solved using eigen but not using inversion.

A: That would be great. This is actually an outstanding research problem that is left open.

Q: I'll let you know if I have an idea! But can we get back for a moment to the connection with first-order logic with aggregates? I now know the restriction to three variables needed for MATLANG, but what about the converse? Does any expression in the three variable fragment of first-order logic with aggregates correspond to an expression in MATLANG?

A: We can show some kind of converse, but we would like to first introduce iterations in MATLANG. This way, a more general connection between MATLANG, extended with iterations, and relational query languages can be made.

Q: OK, let's do iterations first then.

3. ΣMATLANG

3.1 MATLANG + Sum Iterations

A: Let us first describe a very simple form of iteration which suffices to express the 4-clique query. The first thing we do is introduce vector variables $v_1,v_2,\dots$ which will be instantiated with the canonical basis vectors $\mathbf{b}_1,\dots,\mathbf{b}_n$ in $\mathbb{R}^{n\times 1}$, of an appropriate dimension n, and then later we allow for summation over these vector variables.

Q: You mean with $\mathbf{b}_j$ a column vector with all zero entries except for its jth entry that holds the value 1?

A: Right. Now consider a MATLANG expression $v_1^t\cdot X\cdot v_2$ that uses our familiar matrix variable X and two "new" vector variables v_1 and v_2. When v_1 and v_2 are instantiated with basis vectors, say $\mathbf{b}_i$ and $\mathbf{b}_j$, respectively, then can you say what this expression computes when a matrix $\mathbf{A}$ is assigned to X?

Q: That would be simply the entry $\mathbf{A}(i,j)\in\mathbb{R}$ of $\mathbf{A}$?

A: Indeed, now consider again the for-loop program for computing the number of 4-cliques you gave earlier. With the understanding that a basis vector $\mathbf{b}_j$ identifies the jth vertex in a graph, we can easily find four vertices that are all pairwise adjacent. Namely, the expression $e_1(X,v_1,v_2,v_3,v_4)$ defined by

$$v_1^t\cdot X\cdot v_2\cdot v_1^t\cdot X\cdot v_3\cdot v_1^t\cdot X\cdot v_4 \cdot v_2^t\cdot X\cdot v_3\cdot v_2^t\cdot X\cdot v_4\cdot v_3^t\cdot X\cdot v_4$$

serves this purpose. Indeed, when v_1, v_2, v_3 and v_4 are assigned $\mathbf{b}_i$, $\mathbf{b}_j$, $\mathbf{b}_k$ and $\mathbf{b}_\ell$, respectively, and X is assigned an adjacency matrix $\mathbf{A}_G$ of an undirected graph, then this expression returns 1 only when there are edges between all pairs of vertices in $\{i,j,k,\ell\}$. The for-loop program also contained a conditional statement to ensure that the four vertices are different. This can be captured by the expression

$$e_2(v_1,v_2,v_3,v_4):=\prod_{\substack{a,b\in\{1,2,3,4\}\\ a\neq b}}(1-v_a^t\cdot v_b)$$

which evaluates to 1 only when all four vector variables are assigned different basis vectors.

Q: By taking the expression $e(X,v_1,v_2,v_3,v_4):=e_1(X,v_1,v_2,v_3,v_4)\cdot e_2(v_1,v_2,v_3,v_4)$ you then get an indicator function for those 4-tuples of vertices that form a 4-clique?

A: Exactly, we now eliminate the vector variables by summing over all basis vectors. Syntactically, we do this by an expression of the form:

$$\Sigma v_1.\Sigma v_2.\Sigma v_3.\Sigma v_4.e(X,v_1,v_2,v_3,v_4)$$

with the semantics that the summation instantiates each vector variable with all basis vectors, one by one, and by iteratively adding the result of evaluating e on $\mathbf{A}_G$ and the current basis vectors $\mathbf{b}_i,\mathbf{b}_j,\mathbf{b}_k$ and $\mathbf{b}_\ell$. In other words, we sum $e(\mathbf{A}_G,\mathbf{b}_i,\mathbf{b}_j,\mathbf{b}_k,\mathbf{b}_\ell)$ for all possible basis vectors. We then scale by the factor $1/24$, as before.

Q: Nice! Clearly, since we are only adding, the order in which we loop through the basis vectors does not matter.

A: Right. We call the extension of MATLANG with such summations ΣMATLANG [25]. Thus, in this language, expressions of the form $\Sigma v.e$, where e is an expression in ΣMATLANG that may use the vector variable v, are allowed. We will be mostly interested in expressions in ΣMATLANG in which all vector variables are under the scope of a summation. By the way, do you also see how the counting of k-cliques, for any k, can be done in ΣMATLANG?

Q: Sure, you just need enough, k I guess, vector variables and form a similar expression as for the 4-cliques. This also shows that ΣMATLANG is strictly more powerful than MATLANG, if I understood things correctly.

A: Indeed. And in ΣMATLANG we can actually prune some of the operations in MATLANG, such as the ones-vector operation, or the operation which puts a vector on the diagonal of an appropriately sized matrix. These are expressible using summation.

3.2 Expressive Power of ΣMATLANG

Q: What about matrix inversion? Can you do this in ΣMATLANG?

A: Matrix inversion is still out of reach. The reason is because we can translate any ΣMATLANG expression into first-order logic with aggregates, now without the finite variable restriction. The same argument as for MATLANG then shows that matrix inversion would enable ΣMATLANG to compute transitive closure. This is not possible since, as mentioned before, transitive closure is beyond first-order logic with aggregates.

Q: I am now going to press you for an answer to my previous question: Can you do every first-order logic with aggregate query in ΣMATLANG? You promised an answer to this question once you introduced iterations.

A: Sure. Because ΣMATLANG is a procedural language, it allows for a more elegant comparison with the relational algebra with aggregates. Otherwise, similarly as how you relate first-order logic to the relational algebra, safety conditions on logical expressions need to be in place [1], and this becomes quite messy when aggregates are present. And actually, it is even more elegant to compare to the relational algebra over semiring-annotated relations [28, 35].

Q: What's the idea?

A: Instead of explicitly adding aggregations to the relational algebra, one can redefine the standard relational operations on relations that are annotated with semiring values, such that aggregations take place implicitly.

Q: Can you recall what a semiring is?

A: Typical examples of (commutative) semirings are the reals $(\mathbb{R},+,\times,0,1)$, the natural numbers $(\mathbb{N},+,\times,0,1)$, and the boolean semiring $(\{0,1\},\vee,\wedge,0,1)$. More generally, a semiring is just an algebraic structure $\mathbb{S}=(S,\oplus,\otimes,\mathbb{0},\mathbb{1})$ where S is a non-empty set, $\oplus$ and $\otimes$ are binary operations over S, and $\mathbb{0},\mathbb{1}\in S$. We assume that $\oplus$ and $\otimes$ are commutative and associative, $\otimes$ distributes over $\oplus$, and $\mathbb{0}$ and $\mathbb{1}$ are the identities of $\oplus$ and $\otimes$ respectively, and $\mathbb{0}\otimes s=s\otimes\mathbb{0}=\mathbb{0}$.

Q: Ah yes, this rings a bell. Are you going to consider matrices over such semirings?

A: Indeed, you can easily verify that the semantics of expressions in ΣMATLANG naturally lifts when matrices contain semiring values instead of real values. Let us show how semiring-annotated relations represent semiring-valued matrices. Consider a 2×2 matrix $\mathbf{A}$ with elements from a semiring $\mathbb{S}$:

$$\mathbf{A}:=\begin{pmatrix} s_{11} & s_{12} \\ s_{21} & s_{22} \end{pmatrix}$$

One can then represent this matrix as an $\mathbb{S}$-annotated relation, $\mathsf{enc}(\mathbf{A})$, shown below, containing the indices of rows and columns as standard relational tuples, and in which tuple (i,j) is annotated with the corresponding $\mathbb{S}$-value as described by the matrix $\mathbf{A}$:

$\mathsf{enc}(\mathbf{A}):=$

row	col		$\mathbb{S}$
1	1	$\mapsto$	s_{11}
1	2	$\mapsto$	s_{12}
2	1	$\mapsto$	s_{21}
2	2	$\mapsto$	s_{22}

One can now redefine the familiar relational operations on such annotated tables. Intuitively, a join corresponds to applying $\otimes$ on the annotations of the joining tuples, a projection corresponds to applying $\oplus$ on the projected tuples. For example, if we denote by $M(\mathsf{row},\mathsf{col})$ a standard binary relation schema to encode matrices, then the relational algebra expression

$$\pi_{\mathsf{row},\mathsf{col}}(\rho_{\mathsf{col}/\mathsf{col}'}(M)\bowtie\rho_{\mathsf{row}/\mathsf{col}'}(M))$$

evaluates on the $\mathbb{S}$-relation $\mathsf{enc}(\mathbf{A})$ given earlier to:

row	col		$\mathbb{S}$
1	1	$\mapsto$	$s_{11}\otimes s_{11}\oplus s_{12}\otimes s_{21}$
1	2	$\mapsto$	$s_{11}\otimes s_{12}\oplus s_{12}\otimes s_{22}$
2	1	$\mapsto$	$s_{21}\otimes s_{11}\oplus s_{22}\otimes s_{21}$
2	2	$\mapsto$	$s_{21}\otimes s_{12}\oplus s_{22}\otimes s_{22}$

In other words, it represents the matrix product $\mathbf{A}\cdot\mathbf{A}$.

Q: So you implicitly manipulate numbers, or better semiring values, by means of operations on the tuples, which in your case are indices in matrices or vectors?

A: Precisely, and this eliminates the need to syntactically introduce aggregates in the language. This semiring-annotated relational algebra formalisation is quite popular and was proposed by Green et al. in [28]. Now, as it turns out, ΣMATLANG over semiring-valued matrices, is equivalent to the positive relational algebra over semiring-annotated relations, that encode matrices. Here, positive means that the difference operation is not allowed. Of course, the relational algebra expressions are assumed to return a matrix. The correspondence involves a simple translation from ΣMATLANG to the positive relational algebra over semiring-annotated relations, and back [25].

Q: Wait. So, you are basically saying that there was already a matrix query language in the disguise of the positive relational algebra on annotated relations?

A: Yes, indeed. It shows again that it is hard to beat the relational algebra.

We can also tie ΣMATLANG to Functional Aggregate Queries (FAQs) [2, 3]. Indeed, ΣMATLANG expressions correspond to restricted FAQs, using a single semiring and only allowing matrices as input. FAQs more generally work over multiple semirings, do allow for tensor inputs, and are known to capture many computational problems in databases, machine learning and AI. Moreover, a summation in ΣMATLANG corresponds to the elimination of a variable in FAQs, and hence, variable elimination techniques of FAQs [3] can be used to efficiently evaluate ΣMATLANG expressions.

Q: What about MATLANG? That is, when no summation iteration is allowed?

A: Actually, the connection to the positive relational algebra over semiring-annotated relations was made first for MATLANG [16]. Now, it is not that difficult to see that MATLANG is included in ΣMATLANG in which only three vector variables v_1,v_2,v_3 can be used. Conversely, and this is less immediate, one can define a "three-variable" restriction of the positive relational algebra [16] and show that this precisely corresponds to MATLANG or ΣMATLANG with only three vector variables. We believe that this correspondence generalises to the use of k vector variables in ΣMATLANG and a k-variable fragment of the positive relational algebra.

3.3 Beyond Matrices

Q: And what about the full relational algebra over general, not necessarily binary, annotated relations?

A: The focus of our query languages was to work on matrices. As such we are limited to matrix inputs and outputs. So, to deal with inputs that may be of arbitrary arity, one can consider so-called associative tables. For example, $\mathsf{enc}(\mathbf{A})$ would be an associative table of schema $R(\underline{\mathsf{row}},\underline{\mathsf{col}},\mathsf{val})$, with keys underlined, containing elements (i,j,s) with $i,j\in\{1,2\}$ and $s\in\mathbb{S}$. More generally, relations of the form $R(\underline{\mathsf{dim}_1},\underline{\mathsf{dim}_2},\ldots,\underline{\mathsf{dim}_k}, \mathsf{val}_1,\ldots,\mathsf{val}_\ell)$ can be considered.

Q: This would allow for storing tensors, I mean multidimensional arrays, but also standard relations?

A: Precisely. Now, a query language, LARA [31], was proposed for such tables in order to query matrix, tensor and relational data in unison. This language is much alike the relational algebra on semiring-annotated relations but with the support for user-defined functions. It is known that LARA is equivalent to (a safe fragment of) first-order logic augmented with aggregates and relations encoding the user-defined functions [10]. So also here, first-order logic with aggregates is hard to beat.

The LARA language can further be equipped with built-in predicates on key attributes, allowing it to encode the convolution operation, among other things, which is beyond the capabilities of ΣMATLANG [10].

In another recent proposal, a tensor query language, dubbed tensor relational algebra, was considered [45] to deal with multi-dimensional arrays. Its expressive power is not known, but since it includes operations that allow the restructuring of tensors, connections to the nested relational algebra on semiring-annotated relations seem likely.

Finally, the FAQs mentioned earlier gracefully deal with tensor inputs and outputs and are again closely related to the positive relational algebra over semiring-annotated relations of arbitrary arity.

Q: I am actually a little bit disappointed. I mean, what did we learn from all this in terms of linear algebra? Or what to do with these insights? It seems that we only need standard query languages, albeit lifted to semiring-annotated relations?

A: A takeaway message may be that semiring-annotated enabled DBMS's and relational algebra queries over them, already gives you some linear algebra capabilities. That is, everything expressible in ΣMATLANG or by FAQs. And, for doing more complex operations such as matrix inversion, convolution, or eigenvalue-related operations, extensions are needed. A possible approach would again be to investigate the impact of adding specific operations to ΣMATLANG.

Another approach, would be to go beyond the additive updates underlying the iteration in ΣMATLANG. In fact, the need for iteration and limited forms of recursion is also advocated in practical systems [32].

Furthermore, to go beyond aggregate logics, one may argue that a better yardstick for comparison are arithmetic circuits. Indeed, the latter are known to be able to compute most linear algebra operations. So, we can ask whether MATLANG can be extended such that it captures an important class of arithmetic circuits. We actually know how to do that by generalising what kind of computations are allowed when iterating over the vector variables mentioned earlier.

Q: That sounds interesting. I must say that I do not know much about arithmetic circuits. Do you want to talk about this next?

A: We first want to talk a bit more about MATLANG and ΣMATLANG in relation to their distinguishing power. That is, so far we consider expressiveness in an instance-independent way. That is, we wanted to know whether we can find a fixed expression that computes the number of 4-cliques on any given graph, or an expression that computes the matrix inverse for any matrix.

We can, however, also look at instance-dependent expressiveness. In this setting, we can use a varying number of expressions for differentiating matrices.

Q: Not sure where this is heading to, but please go ahead.

4. DISTINGUISHING MATRICES

4.1 Distinguishing Matrices by Queries

A: Let's make this more concrete. We want to understand when, say, two matrices $\mathbf{A}$ and $\mathbf{B}$ are indistinguishable by functions expressible in a matrix query language $\mathcal{L}$. An expressible function corresponds to a "sentence" in $\mathcal{L}$ that, when evaluated on matrices, always returns a scalar. We then say that $\mathbf{A}$ and $\mathbf{B}$ are $\mathcal{L}$-equivalent for a matrix query language $\mathcal{L}$ when $e(\mathbf{A})=e(\mathbf{B})$ for all sentences e in $\mathcal{L}$. We will denote this by $\mathbf{A}\equiv_{\mathcal{L}}\mathbf{B}$.

Q: Why only consider sentences, i.e., scalar functions?

A: Allowing all expressions would result in a too strong notion of equivalence. Indeed, just take $e(X):=X$. Then, $e(\mathbf{A})=e(\mathbf{B})$ requires $\mathbf{A}=\mathbf{B}$ which prevents us from gaining interesting insights.

Q: What extra information can then be gained about matrix query languages when considering the sentence-based notion of equivalence?

A: To illustrate this, let $\mathcal{L}$ be a sub-language of $\mathsf{MATLANG}$ in which we can only do matrix transposition, matrix multiplication, use the ones-vector operation, and turn a vector into a diagonal matrix. We denote this fragment $\mathcal{L}$ by $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$. Then, we may ask whether $\mathbf{A}\equiv_{\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})}\mathbf{B}$ implies that $\mathbf{A}$ and $\mathbf{B}$ are orthogonally similar. That is, whether there exists an orthogonal matrix $\mathbf{O}$ such that $\mathbf{A}=\mathbf{O}\cdot\mathbf{B}\cdot\mathbf{O}^{-1}$ holds. Here, an orthogonal matrix is a matrix satisfying $\mathbf{O}^{-1}=\mathbf{O}^t$. Now, we will use that such a matrix $\mathbf{O}$ exists if and only if $\mathbf{A}$ and $\mathbf{B}$ are co-spectral [39].

Q: Co-spectrality means that these matrices have the same multi-set of eigenvalues, right?

A: Indeed, this turns out not to be the case when sentences are restricted to those in $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$. Now, what about $\mathbf{A}\equiv_{\mathsf{MATLANG}}\mathbf{B}$, does this imply that $\mathbf{A}$ and $\mathbf{B}$ are co-spectral?

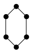
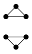
Q: I don't know. You argued before that $\mathsf{MATLANG}$ is not very well-suited for eigenvalue-related operations, so my intuition says no.

A: Well, actually co-spectrality is preserved for matrices that are $\mathsf{MATLANG}$-equivalent. To see this, let us focus on symmetric matrices only. Then, it is known that (symmetric) matrices $\mathbf{A}$ and $\mathbf{B}$ are co-spectral if and only if $\mathsf{tr}(\mathbf{A}^i)=\mathsf{tr}(\mathbf{B}^i)$ for all $i=0,1,\ldots,n-1$. Now, clearly we can compute $\mathbf{A}^i$ for any i in $\mathsf{MATLANG}$.

Q: And you told before how to compute the trace...

A: Precisely, so when $\mathbf{A}\equiv_{\mathsf{MATLANG}}\mathbf{B}$ holds, then also $e(\mathbf{A})=e(\mathbf{B})$ for all sentences e in $\mathsf{MATLANG}$, including the expressions $\mathsf{tr}(X^i)$ for all i. And this implies co-spectrality, as just mentioned.

Q: I see, but why wasn't this true already for $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$? Did you find two matrices that are $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$-equivalent, yet are not co-spectral?

A: Indeed, the adjacency matrices $\mathbf{A}_{G_1}$ and $\mathbf{A}_{H_1}$ of the graphs G_1  and H_1  suffice for this purpose. These are known not to be co-spectral. An easy way to see this is to recall that $\mathsf{tr}(X^i)$ actually computes the number of cycles of length i. Clearly, H_1 contains cycles of length three, whereas G_1 does not. So, $\mathbf{A}_{G_1}$ and $\mathbf{A}_{H_1}$ cannot be co-spectral.

Q: But, how can you tell that they satisfy the same sentences in $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$?

4.2 Connections to Finite Variable Logics

A: To answer this, we observe that G_1 and H_1 are fractionally isomorphic [40]. That is, there exists a non-negative real matrix $\mathbf{S}$ in which each row and each column sums up to one, and such that $\mathbf{A}_{G_1}\cdot\mathbf{S}=\mathbf{S}\cdot\mathbf{A}_{H_1}$ holds. The matrix $\mathbf{S}$ is also called doubly stochastic, and such matrices are a relaxation of permutation matrices. A permutation matrix being such that every row and every column contains at most one non-zero value, which is value 1. Note that when $\mathbf{A}_{G_1}\cdot\mathbf{P}=\mathbf{P}\cdot\mathbf{A}_{H_1}$ holds for a permutation matrix $\mathbf{P}$, then G_1 and H_1 are isomorphic. Hence, the name fractionally isomorphic when considering doubly stochastic matrices. Now, we can show, by induction on expressions e in $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$, that when $\mathbf{A}_{G_1}\cdot\mathbf{S}=\mathbf{S}\cdot\mathbf{A}_{H_1}$ holds, then $e(\mathbf{A}_{G_1})=e(\mathbf{A}_{H_1})$. This provides insights in invariance properties of linear algebra operations included in $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$.

Q: Is the converse also true? Does $\mathbf{A}_G\equiv_{\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})}\mathbf{A}_H$ imply that G and H are fractionally isomorphic?

A: Yes it does. The operations in that fragment suffice to create a collection of sentences $(e_i)_{i\in I}$ such that when $e_i(\mathbf{A}_G)=e_i(\mathbf{A}_H)$ for all $i\in I$, then G and H have a common equitable partition, which in turn is known to imply that G and H are fractionally isomorphic [40]. In addition, this correspondence also implies that $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag})$-equivalence is the same as equivalence of G and H by means of sentences in the two-variable fragment of first-order logic with counting C_2 [17].

Q: If I recall correctly, the logic C_k is first-order logic using k variables only, but in which you can additionally use counting quantifiers of the from $\exists^{\geq t}x_i\varphi$ indicating the existence of at least t distinct elements satisfying φ?

A: Precisely. We can similarly show that $\mathbf{A}_G\equiv_{\mathsf{MATLANG}}\mathbf{A}_H$ if and only if $G\equiv_{\mathsf{C}_3}H$ [24]. So again, note the three variable correspondence. You now see that by considering equivalence of matrices, we can get a fine-grained understanding of the different operations in $\mathsf{MATLANG}$. If you would throw in, e.g., the trace operation, and consider $\mathsf{ML}(\cdot,{}^t,\mathbb{1},\mathsf{diag},\mathsf{tr})$-equivalence, you obtain an equivalence notion on graphs that lies between C_2- and C_3-equivalence. We fully explored the equivalence of adjacency matrices for operations in $\mathsf{MATLANG}$ in [24], and lifted some of the results to arbitrary matrices in [23]. For each sublanguage, one can further identify a corresponding notion of similarity between matrices, by considering other kinds of matrices than doubly stochastic matrices. The more operations we allow, the more restrictive the similarity notion is.

4.3 Beyond MATLANG

Q: What if you extend $\mathsf{MATLANG}$ with matrix inversion?

A: It does not increase distinguishing power. The reason being that the Cayley-Hamilton Theorem [8] states that the inverse of a matrix can be expressed in terms of $\mathsf{MATLANG}$ expressions $\mathsf{tr}(X^i)$ for various i. So, when $\mathbf{A}\equiv_{\mathsf{MATLANG+inv}}\mathbf{B}$ holds, then any sentence that uses the matrix inverse operation inv can be rewritten as a sentence in $\mathsf{MATLANG}$. We emphasise that we can here use sentences that depend on the given matrices. We will need n sentences when the matrices have dimension n. It thus does not contradict our earlier claim that matrix inversion is not expressible in MATLANG by expressions that work for any matrix input.

Q: And C_k-equivalence of graphs for arbitrary k?

A: This turns out to correspond to equivalence of their adjacency matrices in ΣMATLANG in which only k vector variables can be used. We thus complement existing characterisations of C_k-equivalence of graphs, in terms of the k-dimensional Weisfeiler-Leman graph isomorphism test [17], or homomorphism count vectors [20, 22], by one involving functions computable in MATLANG, for $k{=}2$ and $k{=}3$, and in the k-vector variable fragment of ΣMATLANG for arbitrary k.

Q: Sounds all very interesting but I am eager to find out more about the connections to arithmetic circuits. Can we do that next?

5. FOR-MATLANG

5.1 MATLANG + General For Loops

A: Sure, let's do that. Before we get to the technical details I need you to take a step back and do something very basic. Namely, think about classic linear algebra algorithms that you saw when you first learned about matrices.

Q: You mean like Gaussian elimination, LU-factorisation, computing the transitive closure of an adjacency matrix for a graph, …

A: Precisely. Now think about the basic "ingredients" of these algorithms. What are they?

Q: Well, at the most atomic level, I guess you have to be able to access a single entry in a matrix, and also iterate up to the matrix dimension. Using vector variables, you showed already how to access matrix entries and iterate over those in ΣMATLANG, but this does not do the trick, does it? I mean you showed me that transitive closure can not be computed there.

A: Right. The issue with ΣMATLANG is that the result you compute is updated by adding the result of some expression that depends on the basis vector used in the iteration, and you can not access what you computed previously. In order to remedy this, in [25], for-MATLANG was introduced. As the name says, this language extends MATLANG by allowing for-loops; i.e. expressions of the form

$$\texttt{for}\, v, Z\, \textbf{.}\, e.$$

Intuitively, Z is a "new" matrix variable which is iteratively updated according to the expression e, and v gets instantiated by the basis vectors of the appropriate dimension. An important point here is that e can depend on both v and Z.

Q: Wait a moment. So, you still iterate over basis vectors, as before, but the update behaviour is now controlled by means of the variable Z and the expression e? How does this tie to ΣMATLANG?

A: In ΣMATLANG the update mechanism simply corresponds to addition: $\texttt{for}\, v, Z\, \textbf{.}\, Z{+}e$, where e is an expression that may use v but not Z. For example, to compute the vector consisting of all ones we can do $\texttt{for}\, v, Z\, \textbf{.}\, Z{+}v$. In this way, ΣMATLANG is contained in for-MATLANG.

Q: I follow. So in for-MATLANG the crux is that one can now aggregate the results of the iterations with other operations besides sum, like e.g., matrix multiplication?

A: Precisely, and recall that you can use the current result in your computation. In for-MATLANG we can now allow for multiplicative updates by writing, e.g., $\texttt{for}\, v, Z\, \textbf{.}\, Z{\cdot}e$, and even arbitrary expression e using v and Z. For instance, for a matrix variable X, you could do something crazy like $\texttt{for}\, v, Z\, \textbf{.}\, Z{\cdot}\Big(X{+}Z{+}v{\cdot}v^t\Big)$.

Q: Hmm, not sure why someone would do something like that, but I get the intuition. This seems powerful indeed. Though I am still a bit puzzled as to how you actually compute these iterations.

A: Basically, we start with the zero matrix of the appropriate dimension (same as Z), and then start iterating. In the first iteration (when v is instantiated with the first basis vector $\mathbf{b}_1$), the expression e takes $Z{=}\mathbf{0}$ and evaluates. Denote the result of this iteration $\mathbf{A}_1$. For the second iteration ($v{=}\mathbf{b}_2$), we now evaluate e, but instantiate Z with $\mathbf{A}_1$, thus obtaining $\mathbf{A}_2$. In general, iteration $i{+}1$ assigns $\mathbf{A}_i$ to Z when evaluating e (and $\mathbf{b}_{i+1}$ is assigned to v). The result of the expression is then the result of the final iteration. With some tinkering, we can also show that you can start by assigning to Z and arbitrary matrix $\mathbf{A}$ in iteration one instead of the zero matrix. To denote this, we write $\texttt{for}\, v, Z{=}\mathbf{A}\, \textbf{.}\, e$.

Q: I guess that transitive closure is now within reach?

A: Correct. We can compute the transitive closure of a matrix by simulating the Floyd-Warshall algorithm [19]. The following for-MATLANG expression does the trick:

$$\begin{aligned}
&\texttt{for}\, v_3, Z_1{=}\mathbf{A}_G.\;\; Z_1 +\\
&\quad\texttt{for}\, v_1, Z_2.\;\; Z_2 +\\
&\qquad\texttt{for}\, v_2, Z_3.\;\; Z_3 +\\
&\qquad\quad (v_1^t{\cdot}Z_1{\cdot}v_3{\cdot}v_3^t{\cdot}Z_1{\cdot}v_2){\cdot}v_1{\cdot}v_2^t
\end{aligned}$$

Here we basically copy the Floyd-Warshall algorithm using the for-MATLANG syntax. The first vector v_3 will check whether we can go through the vertex k (using $\mathbf{b}_k$) in order to connect vertices i and j (using $\mathbf{b}_i$ and $\mathbf{b}_j$ in v_1 and v_2). The inner expression simply updates the corresponding position in the matrix Z_3, which in turn propagates the results to Z_2 and finally to Z_1, which will have, at the end of the iterations, a non zero entry in a position (i,j) when there is a path from i to j in our graph. Notice that we start with $\mathbf{A}_G$ stored in Z_1.

Q: Right, I now see the use of iterating expressions that depend on these Z variables. One thing still bothers me. Unlike in ΣMATLANG, here the order in which the basis vectors are iterated over actually matters. Can this functionality be used somehow?

A: Indeed, this is a very important point. As you noted, our semantics of `for` expressions assumes that the basis vectors are always accessed in some fixed order. Interestingly, for-MATLANG is powerful enough to make this order information explicit. In particular, we can construct a matrix $\mathbf{S}_{\leq}$, which has the property that $\mathbf{b}_i^t \cdot \mathbf{S}_{\leq} \cdot \mathbf{b}_j$ equals one if and only if $i \leq j$. With this matrix we can construct all sorts of order operators, allowing us to check whether we are manipulating the first/last basis vector, or to produce the following/previous basis vector from the current one.

5.2 Gaussian Elimination and Matrix Inversion

Q: I guess that the availability of such order information adds considerable power. Can you do Gaussian elimination, or find the inverse of a matrix using this?

A: Actually you can. In fact, in [25], explicit expressions for performing Gaussian elimination, both without and with pivoting, are given. To provide some intuition, think how Gaussian elimination works in the case when no pivoting is required (that is, the diagonal element we are processing in each iteration is non zero).

When working with the matrix $\mathbf{A}$, you take the first diagonal entry $\mathbf{A}(1,1)$, and then iterate over the rows j below this first row, multiplying them by the constant $-\frac{\mathbf{A}(j,1)}{\mathbf{A}(1,1)}$, and adding this row to the previous values. This way, we get all zeros in the first column below $\mathbf{A}(1,1)$, and the process continues.

We can simulate this in for-MATLANG with three nested for-loops: the first one to mark the iteration of the Gaussian elimination; the second one to fetch the rows that follow; and the final one to update the entries of each row fetched by the second for-loop. Having explicit order via the matrix $\mathbf{S}_{\leq}$ allows us to check when the basis vector in the second for-loop corresponds to a row that should be modified in this iteration. The computation that needs to be performed requires some details though, and can be found in [25]. In essence, it boils down to defining elementary matrices to perform row transformations.

Q: But you need the division function for this, correct?

A: Precisely. In order to perform the reduction of the current column we need to divide with the diagonal element. But you would probably be hard pressed to come up with an algorithm that can do Gaussian elimination without being able to divide two numbers.

Q: I'll give you that. What about pivoting?

A: Well, if you allow us to also compare whether a number is greater than 0 we can perform a so-called PLU factorisation as well. Namely, we can construct expressions $e_{L^{-1}\cdot P}$ and e_U such that evaluating them on the matrix $\mathbf{A}$ as the input, they give matrices $\mathbf{L}^{-1} \cdot \mathbf{P}$ and $\mathbf{U}$ such that $\mathbf{L}^{-1} \cdot \mathbf{P} \cdot \mathbf{A} = \mathbf{U}$, where $\mathbf{P}$ is a permutation matrix. The greater than zero function here is used to simulate a limited if-then-else operation, which allows us to do pivoting.

Q: And can you push this all the way to compute inverses and determinants?

A: Sure, one can construct expressions e_{det} and e_{inv} that return the determinant of a matrix $\mathbf{A}$, and its inverse matrix, respectively, whenever A is invertible [25]. When $\mathbf{A}$ is not invertible, these expressions simply return the zero matrix. Actually the greater than zero function can be eliminated here, as our connection with arithmetic circuits will show.

5.3 Linear Algebra and Arithmetic Circuits

Q: Right. Let's take a step back. You initially devised MATLANG in order to gain an understanding of the expressive power of linear algebra operations. Now, with for-MATLANG you have a quite powerful iteration mechanism at hand which allows you to do complex linear algebra computations. How can we gain insights in the power of specific linear algebra operations? For example, what does this all imply for transitive closure or matrix inversion? Furthermore, what is the limit of for-MATLANG in terms of expressive power?

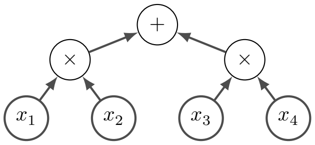
A: That are a lot of questions. To better understand specific linear algebra operations, one way that this can be approached is by investigating what kind of update mechanisms are needed in the for-loops. More precisely, how complex does the expression e in $\mathtt{for}\, v, Z\, \mathtt{.}\, e$ needs to be, in order to do, say transitive closure. We actually looked into that, but let's get back to this later on. To answer your question about the limit of for-MATLANG, we will compare with arithmetic circuits.

Q: OK, let's move on to circuits first then. But tell me one thing first: why compare with arithmetic circuits?

A: If you recall, our objective was to have a rough approximation of what a core language for linear algebra should be. And arithmetic circuits of bounded depth are sometimes said to capture linear algebra [4]. In a broad sense of course. The intuition is that arithmetic circuits perform basic number manipulations, and pass them up, as one does when, for example, multiplying matrices. Also, most standard linear algebra algorithms (such as the ones we discussed thus far), can be expressed by arithmetic circuits [41, 42].

Q: I guess I could sort of see your point. Or rather, if I press you further, you will ask me what is a good formalism that captures linear algebra, and I don't have a definitive answer. So let us proceed. Can you please explain first what arithmetic circuits are?

A: Quite simple: they are similar to boolean circuits [7], but the input gates now take numbers over $\mathbb{R}$, and the computations they perform are $+$ and $\times$, not just boolean operations. For instance, $f(x_1,x_2,x_3,x_4):= x_1\times x_2+x_3\times x_4$, can be computed with the circuit:



Basically, an arithmetic circuit is a directed acyclic graph, where nodes are called gates, and the edges wires. Gates denoted by variables (e.g. x_3 above) have no incoming arrows, and are called input gates. If a gate has no outgoing arrows it is called an output gate. When a circuit ϕ has n input gates and a single output gate, it computes a polynomial function $f:\mathbb{R}^n\mapsto\mathbb{R}$, in a natural way. In this case, we will write $\phi(a_1,\ldots,a_n)$ for the number computed by the output gate when receiving $a_1,\ldots,a_n$ in the input gates.

5.4 FOR-MATLANG and Arithmetic Circuits

Q: Since for-MATLANG expressions can be evaluated on matrix inputs of arbitrarily large dimensions, to compare for-MATLANG with arithmetic circuits, I guess you need a notion of uniformity to handle circuits with a varying number of input gates?

A: Precisely. As usual, we will work with a circuit family $\{\phi_n\,|\,n=1,2,\ldots\}$, where each ϕ_n has n inputs and a single output. We will call this family uniform, if there is a LOGSPACE Turing machine, which, given 1^n as input, produces a description of ϕ_n. Finally, we will need to restrict our circuit families a bit. For now, we will focus of families of logarithmic depth, where the depth is defined as the longest distance from any input gate, to the output gate of the circuit.

In [25] we then show that for any uniform family $\{\phi_n\,|\, n=1,2,\ldots\}$, where depth of ϕ_n is bounded by $\mathcal{O}(\log(n))$, we have a single for-MATLANG formula e_ϕ, which uses a vector variable v, such that for any k and $a_1,\ldots,a_k\in \mathbb{R}$, it holds that evaluating e_ϕ by assigning the vector $[a_1\cdots a_k]^t$ to v, produces the same result as computing $\phi_k(a_1,\ldots,a_k)$. Effectively, we can simulate any log-depth uniform circuit family with a single for-MATLANG formula! So, for-MATLANG is quite powerful.

Q: I share your enthusiasm, but there seems to be a lot going on here. Can we please unpack this a little bit? First, how does the Turing machine for generating the circuit interact with your for-MATLANG expression?

A: Actually, here we show something more general: any Turing machine which works in polynomial time, uses only linear space on its work tape, and produces a linear size output, can be simulated by means of a for-MATLANG expression. In particular, this allows us to have an expression e_M, which simulates the LOGSPACE machine M that generates the circuit family.

Q: Hmm, I'll read the paper for more details. Can you say more about the polynomial functions computed by log-depth uniform circuit families and thus by for-MATLANG? For example, what about the degree of the polynomials that your circuits compute? Can any connection be drawn there?

A: Indeed. First, given that each circuit ϕ_n with n inputs and a single output computes a polynomial over $\mathbb{R}^n$, we will call the degree of the circuit ϕ_n the degree of this polynomial. We will say that a family of arithmetic circuits $\{\phi_n\,|\,n=1,2,\ldots\}$ is of polynomial degree, if there is a polynomial p such that the degree of ϕ_n is bounded by $p(n)$. Our previous results only tells us that a polynomial degree family of circuits can be simulated in for-MATLANG when the family is also of logarithmic depth.

We can, however, actually drop the restriction on circuit depth due to the result of Valiant et. al. [44] and Allender et. al. [5] which says that any function computed by a uniform circuit family of polynomial degree (and polynomial depth), can also be computed by a uniform circuit family of logarithmic depth. Therefore, for-MATLANG can simulate any polynomially bounded uniformly generated circuit family. We can also extend these result to circuits with multiple output gates. The degree here is simply the maximum of the degrees of all the output gates.

Q: It seems fairly easy to come up with a for-MATLANG expression that computes a polynomial of exponential degree. For example, consider the following for-loop program $\mathsf{exp}(x,n)$ given by

```
for $i=1,\ldots,n$ do
    $x=xx$
return $x$
```

would compute $\mathsf{exp}(a,n)=a^{2^n}$ for $a\in\mathbb{R}$. Clearly not polynomial. I assume that $\mathsf{exp}(x,n)$ can be encoded in for-MATLANG? So, can circuits simulate for-MATLANG? Or did you go too far?

A: Yes, $\mathsf{exp}(x,n)$ can be encoded in for-MATLANG. And no, we did not go too far. If we extend arithmetic circuits in a natural way to take matrices as inputs (by arranging the indices into input gates), and produce matrices as output (by allowing multiple output gates), we can also show that for any for-MATLANG expression e, there is a uniform circuit family $\{\phi_n^e\}$, computing the same function over matrices.

Q: Aren't you hiding something here? The depth versus degree connection seems lost with this result?

A: Yes, you are correct. When we go from for-MATLANG to circuits we put no restrictions on the degree of the circuits, in this way we can take care of your $\mathsf{exp}(x,n)$ program. For the other way around, we need polynomial degree circuits.

To save our face we can offer the following equivalence: consider any function f that as its input takes matrices $\mathbf{A}_1,\ldots,\mathbf{A}_k$, and produces a matrix $\mathbf{A}$ as output. The dimension of each $\mathbf{A}_i$ is $\alpha_i\times\beta_i$, with $\alpha_i,\beta_i\in\{n,1\}$. That is, the function works on matrices of different dimensions for each n, same as for-MATLANG, or circuit families. Then we can show that f is computed by a polynomial degree uniform circuit family if and only if it is computed by a for-MATLANG formula of polynomial degree. Here a for-MATLANG formula is of polynomial degree if it has an equivalent polynomial degree circuit family.

Q: OK, so the equivalence holds for circuits and expressions of polynomial degree. Can you detect when your formulas are of polynomial degree easily?

A: Actually we can not. That is, the problem of determining whether a for-MATLANG expression is of polynomial degree is undecidable [25]. This result follows from the fact that we can simulate Turing machines (up to a fixed amount of space).

Q: And do you at least know some nice class of expressions in for-MATLANG of polynomial degree?

A: We actually already showed you one: ΣMATLANG.

Q: Yes, but for that fragment we did not really need the detour to circuits, right? I now notice that your circuits only use sum and product. Given that algorithms such as Gaussian elimination need to perform division, I find it difficult to believe circuits can simulate this. Can you please explain?

A: OK, you caught us red handed again. But we did not lie actually. Basically, the arithmetic circuit people did all of our work for us [4]. What they showed is that in circuits that use sum, product, and division, the division operator can be postponed as to be used only once at the output gate without affecting the result [43, 14, 34].

So we are left with circuits that use only sum and product (the degrees also remain polynomial), and can use division at the very end. What we can then show is that a function over matrices is computed by a uniform family of polynomial degree that uses sum, product and division, if and only if it is computed by a for-MATLANG expression of polynomial degree that uses division.

Q: I see. I take away from all this that for-MATLANG is bounded by uniform circuit families, and that a large class of uniform circuits can be simulated in for-MATLANG. I have an online lecture soon, but before I go, can we get back to my earlier question about what kind of update mechanism are needed to carry out specific linear algebra computations, such as transitive closure or matrix inversion?

6. FRAGMENTS OF FOR-MATLANG

A: Do you remember how we defined the sum iteration of ΣMATLANG with the for-operator?

Q: Sure. It was something like $\mathtt{for}\, v,Z\,.\,Z+e$, where e is an expression that could use v but not Z.

A: Exactly. This expression stores the partial outputs in Z and updates them by summing with e and the next basis vector. There is nothing special about the use of the sum here. We can use the same expression but with a different operator like:

$$\mathtt{for}\, v,Z{=}\mathbf{I}_\odot\,.\,Z\odot e$$

where $\odot$ is any operation between matrices and $\mathbf{I}_\odot$ initialise Z accordingly to $\odot$.

Q: I see where you are going. You want to define new operators included in for-MATLANG in analogy to how ΣMATLANG was defined.

A: Yes, there we go. Now, for-MATLANG gives a language for computing linear algebra with the same expressive power as arithmetic circuits. We can restrict the for-loops with the above trick to understand, for example, which restrictions capture transitive closure or matrix inversion.

Q: I like the idea. A natural choice for $\odot$ is matrix multiplication, but the output will be zero if the for-loop starts with the zero matrix. Not very interesting.

A: You are right, and for this reason, we need the initialisation matrix $\mathbf{I}_\odot$. For matrix multiplication, we can use the identity matrix. Another choice is to use the Hadamard product for $\odot$ and initialise the loop with the the matrix $\mathbf{I}_\odot$ having value one in all its entries.

Q: Hadamard product? Sorry, I am not familiar with this operation.

A: Hadamard product is the pointwise multiplication between matrices of the same dimension. If we use this product in place of $\odot$ in the above expression, we get an iteration operator which we denote by $\Pi^{\mathrm{H}}v.e$. It takes the pointwise product of evaluating e with v replaced by basis vectors. Then we define ΣPMATLANG (sum-product MATLANG) as the extension of ΣMATLANG with this operator. Since the Hadamard product is commutative, the order used to iterate over basis vectors does not matter here, just as for ΣMATLANG.

Q: I agree that it is a natural next step after ΣMATLANG. So, what can you do with such an operator? Does it extend the expressive power of ΣMATLANG?

A: Indeed. For example, you can compute the product of the values in the diagonal of a matrix $\mathbf{A}$ by evaluating $\Pi^{\mathrm{H}} v.v^t \cdot X \cdot v$. Its output grows exponentially with the matrix dimension, but the output of a ΣMATLANG formula is bounded by a polynomial. So, ΣPMATLANG is strictly more expressive than ΣMATLANG.

Q: OK, this is a nice way to extend ΣMATLANG. You already told me that ΣMATLANG coincides with positive relational algebra over semiring-annotated relations. Can you get a similar equivalence for this fragment?

A: Yes, but we need to move from relational algebra to another logic formalism over annotated relations, called weighted logic [21]. Roughly speaking, weighted logic extends first-order logic from the boolean semiring to any semiring, using the sum instead of disjunction and product instead of conjunction. In particular, $\exists x$ and $\forall x$ are evaluated as sums and products over the domain. This logic was used for characterising the expressive power of weighted automata [21], but recently it has been proposed as a first-order logic for provenance [27] and for studying the descriptive complexity of counting complexity classes [6].

Similarly to ΣMATLANG and positive relational algebra, we can show that ΣPMATLANG has the same expressive power as weighted logics over annotated binary relations [25]. There is a natural correspondence between the sum and product (Π^H) operator and existential and universal quantification of weighted logic.

Furthermore, connections can be drawn again with the FAQs mentioned earlier. Indeed, FAQs support summation and multiplication over variables and in this way also extend existential and universal quantification over semiring annotated relations.

Q: Nice connection! But if ΣPMATLANG coincides with first-order logic, I guess it cannot express transitive closure. Let's say that I am a bit disappointed. But, wait, we haven't discussed yet what happens if I use matrix multiplication for $\odot$ instead of the Hadamard product.

A: Yes, if we use matrix multiplication for $\odot$, we can get an iteration operator we which denote by $\Pi v.e$. We call prod-MATLANG the extension of ΣMATLANG with this operator. One can show that $\Pi^{\mathrm{H}} v.e$ can be simulated with $\Pi v.e$, and thus ΣPMATLANG is included in prod-MATLANG. Expressions in prod-MATLANG are no longer invariant under a change of order among the basis vectors, due to the use of matrix multiplication, by contrast to expressions in ΣPMATLANG.

Q: I am a bit lost in all these fragments, but I see that you have a hierarchy of languages inside for-MATLANG. I would expect that prod-MATLANG is strictly more expressive than ΣPMATLANG. Furthermore, by iterating the matrix product, can you do some sort of transitive closure?

A: Indeed, in prod-MATLANG we can define transitive closure. If we add order information in the form of $\mathbf{S}_{\leq}$, we can compute the determinant of a matrix and do matrix inversion. So, this operator is quite powerful. It is an open problem whether this fragment is strictly included in for-MATLANG.

Q: So, we finally reached a fragment that naturally includes transitive closure and matrix inversion. Just in time, because my online lecture is about to start. Before leaving, can you say what remains to be done?

7. CONCLUSION

A: Sure. There are some open problems that can be addressed. For example, we do not know any graph property expressible in MATLANG+eigen but not in MATLANG+inv. Also, understanding MATLANG extended with complex linear algebra operations, such as LU decomposition, Singular Value decomposition, etc., is open. Similarly, the precise expressive power of the various fragments of for-MATLANG mentioned above is unknown. One may want to compare these fragments to various recursive extensions of first-order logic.

Other iteration mechanisms could be considered as well. For example, one may consider a "simple iteration" fragment of for-MATLANG in which an ordinary MATLANG expression is iterated n times, where n is the dimension of the input matrix. This is similar to the for-extension of relational algebra proposed by Chandra [18]. Such expressions are order-independent. Is this strictly weaker than order-independent for-MATLANG? Whereas transitive closure can be computed in this way, it seems unlikely that such simple iterations suffice for matrix inversion.

Furthermore, since many (graph) neural network formalisms are phrased in linear algebra terms, one can view these naturally as expressions in a matrix query language. What do our expressiveness results imply for such machine learning methods? In this context, MATLANG was recently used to extend graph neural networks [9]. If you are interested in this kind of questions, we recommend [29] and [11] (also a principles column) as further reading.

Finally, beyond expressiveness one may look into algorithmic aspects of matrix query languages. In particular, the mentioned connections to Functional Aggregate Queries (FAQs) [2, 3] and their evaluation methods could perhaps be leveraged.

Q: All very interesting. Thanks for introducing me to the world of matrix query languages!

A: Thanks for your questions and insightful comments, it was entirely our pleasure to talk to you.

8. REFERENCES

[1] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[2] Mahmoud Abo Khamis, Ryan R. Curtin, Benjamin Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. On functional aggregate queries with additive inequalities. In *Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, page 414–431, 2019.

[3] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. Faq: Questions asked frequently. In *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, page 13–28. ACM, 2016.

[4] Eric Allender. Arithmetic circuits and counting complexity classes. *Complexity of Computations and Proofs, Quaderni di Matematica*, 13:33–72, 2004.

[5] Eric Allender, Jia Jiao, Meena Mahajan, and V. Vinay. Non-commutative arithmetic circuits: Depth reduction and size lower bounds. *Theor. Comput. Sci.*, 209(1-2):47–86, 1998.

[6] Marcelo Arenas, Martin Muñoz, and Cristian Riveros. Descriptive complexity for counting complexity classes. *Log. Methods Comput. Sci.*, 16(1), 2020.

[7] Sanjeev Arora and Boaz Barak. Complexity theory: A modern approach, 2009.

[8] Sheldon Jay Axler. *Linear algebra done right*, volume 2. Springer, 1997.

[9] Muhammet Balcilar, Pierre Héroux, Benoit Gaüzère, Pascal Vasseur, Sébastien Adam, and Paul Honeine. Breaking the limits of message passing graph neural networks. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pages 599–608. PMLR, 2021.

[10] Pablo Barceló, Nelson Higuera, Jorge Pérez, and Bernardo Subercaseaux. On the expressiveness of LARA: A unified language for linear and relational algebra. In *Proceedings of the 23rd International Conference on Database Theory (ICDT)*, volume 155 of *LIPIcs*, pages 6:1–6:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[11] Pablo Barceló, Egor V. Kostylev, Mikaël Monet, Jorge Pérez, Juan L. Reutter, and Juan-Pablo Silva. The expressive power of graph neural networks as a query language. *SIGMOD Rec.*, 49(2):6–17, 2020.

[12] Peter Baumann, Andreas Dehmel, Paula Furtado, Roland Ritsch, and Norbert Widmann. The multidimensional database system RasDaMan. *SIGMOD Rec.*, 27(2):575–577, June 1998.

[13] Matthias Boehm, Arun Kumar, and Jun Yang. *Data Management in Machine Learning Systems*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2019.

[14] Allan Borodin, Joachim von zur Gathem, and John Hopcroft. Fast parallel matrix and GCD computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 65–71. IEEE, 1982.

[15] Robert Brijder, Floris Geerts, Jan Van Den Bussche, and Timmy Weerwag. On the expressive power of query languages for matrices. *ACM Trans. Database Syst.*, 44(4), 2019.

[16] Robert Brijder, Marc Gyssens, and Jan Van den Bussche. On matrices and K-relations. In *Proceedings of the 11th International Symposium on Foundations of Information and Knowledge Systems (FoIKS)*, volume 12012 of *Lecture Notes in Computer Science*, pages 42–57. Springer, 2020.

[17] Jin-yi Cai, Martin Fürer, and Neil Immerman. An optimal lower bound on the number of variables for graph identifications. *Comb.*, 12(4):389–410, 1992.

[18] Ashok K. Chandra. Programming primitives for database languages. In *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming Languages (POPL*, pages 50–62, 1981.

[19] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[20] Holger Dell, Martin Grohe, and Gaurav Rattan. Lovász meets weisfeiler and leman. In *Proceedings of the 45th International Colloquium on Automata, Languages, and Programming (ICALP)*, volume 107 of *LIPIcs*, pages 40:1–40:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018.

[21] Manfred Droste and Paul Gastin. Weighted automata and weighted logics. In *Proceedings of the 32nd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 3580 of *Lecture Notes in Computer Science*, pages 513–525, 2005.

[22] Zdenek Dvorák. On recognizing graphs by numbers of homomorphisms. *J. Graph Theory*, 64(4):330–342, 2010.

[23] Floris Geerts. When can matrix query languages

discern matrices? In *Proceedings of the 23rd International Conference on Database Theory (ICDT)*, volume 155 of *LIPIcs*, pages 12:1–12:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[24] Floris Geerts. On the expressive power of linear algebra on graphs. *Theory Comput. Syst.*, 65(1):179–239, 2021.

[25] Floris Geerts, Thomas Muñoz, Cristian Riveros, and Domagoj Vrgoč. Expressive power of linear algebra query languages. In *Proceedings of the 40th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, 2021.

[26] Gene H. Golub and Charles F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, fourth edition, 2013.

[27] Erich Grädel and Val Tannen. Semiring provenance for first-order model checking, 2017. http://arxiv.org/abs/1712.01980.

[28] Todd J. Green, Gregory Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the Twenty-Sixth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 31–40. ACM, 2007.

[29] Martin Grohe. Word2vec, node2vec, graph2vec, x2vec: Towards a theory of vector embeddings of structured data. In *Proceedings of the 39th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems (PODS)*, page 1–16. ACM, 2020.

[30] Lauri Hella, Leonid Libkin, Juha Nurmonen, and Limsoon Wong. Logics with aggregate operators. *J. ACM*, 48(4):880–907, 2001.

[31] Dylan Hutchison, Bill Howe, and Dan Suciu. Laradb: A minimalist kernel for linear and relational algebra computation. In *Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond (BeyondMR)*. ACM, 2017.

[32] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. Declarative recursive computation on an RDBMS: Or, why you should use a database for distributed machine learning. *Proc. VLDB Endow.*, 12(7):822–835, 2019.

[33] Dimitrije Jankov, Shangyu Luo, Binhang Yuan, Zhuhua Cai, Jia Zou, Chris Jermaine, and Zekai J. Gao. Declarative recursive computation on an rdbms: Or, why you should use a database for distributed machine learning. *SIGMOD Rec.*, 49(1):43–50, 2020.

[34] Erich Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *Journal of the ACM (JACM)*, 35(1):231–264, 1988.

[35] Grigoris Karvounarakis and Todd J. Green. Semiring-annotated data: Queries and provenance? *SIGMOD Rec.*, 41(3):5–14, 2012.

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

[37] Shangyu Luo, Zekai J. Gao, Michael N. Gubanov, Luis Leopoldo Perez, Dimitrije Jankov, and Christopher M. Jermaine. Scalable linear algebra on a relational database system. *Commun. ACM*, 63(8):93–101, 2020.

[38] Martin Otto. *Bounded Variable Logics and Counting: A Study in Finite Models*, volume 9 of *Lecture Notes in Logic*. Cambridge University Press, 2017.

[39] Sam Perlis. *Theory of matrices*. Addison-Wesley Press, 1952.

[40] Motakuri V. Ramana, Edward R. Scheinerman, and Daniel Ullman. Fractional isomorphism of graphs. *Discrete Mathematics*, 132(1-3):247–265, 1994.

[41] Ran Raz. On the complexity of matrix product. *SIAM J. Comput.*, 32(5):1356–1369, 2003.

[42] Amir Shpilka and Amir Yehudayoff. Arithmetic circuits: A survey of recent results and open questions. *Foundations and Trends in Theoretical Computer Science*, 5(3-4):207–388, 2010.

[43] Volker Strassen. Vermeidung von divisionen. *Journal für die reine und angewandte Mathematik*, 264:184–202, 1973.

[44] Leslie G Valiant and Sven Skyum. Fast parallel computation of polynomials using few processors. In *International Symposium on Mathematical Foundations of Computer Science (MFCS)*, pages 132–139. Springer, 1981.

[45] Binhang Yuan, Dimitrije Jankov, Jia Zou, Yuxin Tang, Daniel Bourgeois, and Chris Jermaine. Tensor relational algebra for machine learning system design. *Proc. VLDB Endow.*, 14(8):1338–1350, 2021.

[46] Ying Zhang, Martin Kersten, and Stefan Manegold. Sciql: Array data processing inside an RDBMS. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, page 1049–1052. ACM, 2013.