A Demonstration of MAGiQ: Matrix Algebra Approach for Solving RDF Graph Queries

Fuad Jamour, Ibrahim Abdelaziz, Panos Kalnis
King Abdullah University of Science and Technology (KAUST)
{fuad.jamour, ibrahim.abdelaziz, panos.kalnis}@kaust.edu.sa

ABSTRACT

Existing RDF engines follow one of two design paradigms: relational or graph-based. Such engines are typically designed for specific hardware architectures, mainly CPUs, and are not easily portable to new architectures. Porting an existing engine to a different architecture (e.g., many-core architectures) entails almost redesign from scratch. We explore sparse matrix algebra as a third paradigm for designing a portable, scalable, and efficient RDF engine. We demonstrate MAGiQ; a matrix algebra approach for evaluating complex SPARQL queries over large RDF datasets. MAGiQ represents an RDF graph as a sparse matrix, and translates SPARQL queries to matrix algebra programs. MAGiQ takes advantage of the existing rich software infrastructure for processing sparse matrices, optimized for many architectures (e.g., CPUs, GPUs, distributed), effortlessly. This demo motivates the adoption of matrix algebra in RDF graph processing by showing MAGiQ's performance with different matrix algebra backend engines. MAGiQ, using a GPU, is orders of magnitude faster in solving complex queries on a billion edge graph than state-of-the-art RDF systems.

PVLDB Reference Format:
Fuad Jamour, Ibrahim Abdelaziz, and Panos Kalnis. A Demonstration of MAGiQ: Matrix Algebra Approach for Solving RDF Graph Queries. *PVLDB*, 11 (12): 1978-1981, 2018.
DOI: https://doi.org/10.14778/3229863.3236239

1. INTRODUCTION

RDF [3] data is a collection of triples of the form ⟨`subject`, *predicate*, `object`⟩ where the predicate describes the relationship between the subject and the object. An RDF dataset can be viewed as a directed edge-labelled graph where each triple corresponds to an edge. The RDF data model has been gaining popularity in various application domains such as the semantic web, bioinformatics, and knowledge graphs [7, 9]. SPARQL is the de-facto query language for RDF data which offers graph pattern matching semantics [16].

Proceedings of the VLDB Endowment, Vol. 11, No. 12
Copyright 2018 VLDB Endowment 2150-8097/18/8.
DOI: https://doi.org/10.14778/3229863.3236239

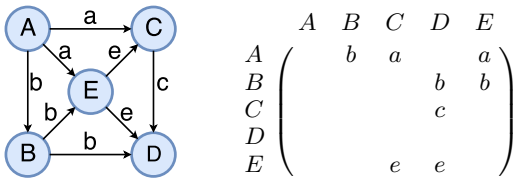

Figure 1: Example RDF graph (left), and corresponding RDF sparse matrix (right).

Many research efforts focus on scalable engines for SPARQL queries over large RDF datasets [7, 16]. Two design paradigms are dominant: the relational paradigm and the graph-based paradigm. The relational paradigm builds exhaustive indices and utilizes relational operators (e.g., joins) to solve SPARQL queries [15, 12, 13]. The graph-based paradigm represents the RDF data in its native graph form and uses graph traversal for query evaluation [8, 18]. Solutions that follow the existing paradigms are designed with a particular hardware architecture in mind, and thus are not easily portable to new architectures. Most existing RDF engines [13, 12, 15, 18] use CPUs. Adapting these engines to run effectively on GPUs, for example, entails (almost) redesign from scratch even though the underlying ideas for query planning and execution are similar.

The development of efficient data structures and algorithms for sparse matrices encouraged many researchers to adopt the matrix algebra formulation for graph problems [14]. GraphBLAS [1] emerged as a convergence of efforts towards building a standard set of sparse matrix algebra primitives for solving graph problems. One of the premises of GraphBLAS is to identify a limited set of operations (e.g., sparse matrix multiplication) that can be used to formulate a wide range of graph algorithms, and build very optimized implementations of these primitives across different architectures. This direction reduces the replication of effort inherent in current scalable graph processing schemes, including RDF graph processing. Many experimental implementations of the GraphBLAS standard are available [10], and a high performance full implementation became available recently; SuiteSparse:GraphBLAS [4].

Motivated by the potential of matrix algebra in graph processing, we demonstrate MAGiQ; a matrix algebra based solution for evaluating SPARQL queries over large RDF graphs. MAGiQ stores an RDF graph as a sparse integer matrix (Figure 1), and translates conjunctive SPARQL queries to concise matrix algebra programs that operate on the matrix representation of the RDF graph. The matrix algebra program produced by MAGiQ query translator consists of a sequence of sparse matrix-matrix multiplications

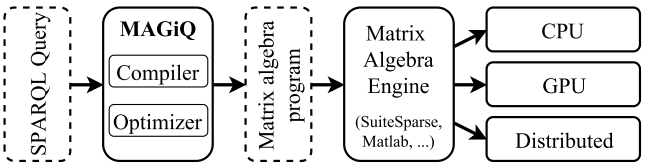

Figure 2: Architecture of MAGiQ

that ultimately compute a collection of sparse matrices that capture the result set of an input SPARQL query. MAGiQ's utilization of matrix algebra makes it portable, scalable, and efficient all together unlike existing RDF engines.

The conference audience will be able to interact with MAGiQ through a graphical interface, where they can select a dataset and type a SPARQL query. The interface will visualize the steps of translating the query to a matrix algebra program. The interface will also display the concise program in MATLAB language. The audience will be able to select a backend (CPU or GPU) and see a comparison between the runtimes of MAGiQ and state-of-the-art specialized RDF engines.

2. OVERVIEW OF MAGiQ

Figure 2 shows the high-level architecture of MAGiQ. The query compiler translates SPARQL queries to matrix algebra programs. The optimizer takes advantage of matrix algebra properties to re-order the operations in a way that produces more efficient programs. Once a matrix algebra program is available, an existing sparse matrix algebra engine such as MATLAB or SuiteSparse:GraphBLAS [4] can be used to evaluate the query over different hardware architectures. Consider the example SPARQL query in Figure 3. This query is translated to the following matrix algebra program (using MATLAB notation), where query edges are processed in the following order: ⟨`?x`, a, `?y`⟩, ⟨`?y`, c, `?z`⟩ and ⟨`?x`, b, `?w`⟩:

$$\mathbf{M}_{xy} = \mathbf{I} * a \otimes \mathbf{A}$$
$$\mathbf{M}_{yz} = \texttt{diag}(\texttt{any}(\mathbf{M}'_{xy})) * c \otimes \mathbf{A}$$
$$\mathbf{M}_{xy} = \mathbf{M}_{xy} \times \texttt{diag}(\texttt{any}(\mathbf{M}_{yz}))$$
$$\mathbf{M}_{xw} = \texttt{diag}(\texttt{any}(\mathbf{M}_{xy})) * b \otimes \mathbf{A}$$

The $\otimes$ symbol denotes matrix multiplication over a semiring, which is explained in Section 2.2. We explain query translation in Section 2.3. The example program above works as follows. The first line selects the valid bindings of variables x and y using predicate a from the RDF matrix $\mathbf{A}$, and stores the results in matrix $\mathbf{M}_{xy}$. The second line uses the bindings of y and predicate c to select the bindings of z. The third line updates the bindings of x and y to eliminate bindings invalidated by predicate c. Finally, the fourth line uses the bindings of x in $\mathbf{M}_{xy}$ with predicate b to select the valid bindings of w. The rest of this section briefly describes the main ideas used in MAGiQ.

2.1 RDF Graph Representation

MAGiQ stores an RDF graph as a sparse square matrix $\mathbf{A} : \mathbb{Z}^{n \times n}$, where n is the number of nodes in the RDF graph (i.e., the number of unique subjects and objects). A non-zero entry at (i, j) with value p_{ij} (i.e., $\mathbf{A}(i, j) = p_{ij}$) means that subject i is connected to object j with predicate p_{ij}. A row $\mathbf{A}(i, :)$ stores predicates of the outgoing edges of node i. Figure 1 shows an example RDF graph with 5 nodes

```
SELECT ?x ?y ?z ?w WHERE {
  ?x <a> ?y .
  ?y <c> ?z .
  ?x <b> ?w .
}
```

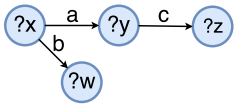

Figure 3: Example SPARQL query (left), and its graph representation (right).

$A, B, \ldots E$ and 5 unique predicates $a, b, \ldots e$. The example graph has 8 triples, which result in 8 non-zero entries in $\mathbf{A}$.

2.2 Selection Matrix and Selection Operation

A *selection matrix* is a diagonal matrix with ones on diagonal entries with row/column indices to be selected. When a selection matrix is multiplied with a matrix of the same size, the product is a matrix with the specified rows/columns present. We refer to the multiplication of a matrix $\mathbf{M}$ with a selection matrix $\mathbf{S}$ as *selection operation*. The following example demonstrates using the selection operation to select a row from a matrix. Let a selection matrix $\mathbf{S}$ have a single one at index $(2, 2)$. As shown below, this row selection operation results in a matrix $\mathbf{C}$ with row 2 present only. The same operation can extract multiple rows by placing more ones at the diagonal entries of $\mathbf{S}$.

$$\underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{S}} \times \underbrace{\begin{pmatrix} a & b & c \\ a & b & c \\ a & b & c \end{pmatrix}}_{\mathbf{M}} = \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ a & b & c \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{C}}$$

A *semiring* is a set with two binary operators; 'addition' and 'multiplication' [14]. A matrix algebra can be defined over many semirings other than the standard arithmetic addition and multiplication. We use a semiring with the set of integers and logical OR as the 'addition' operator and `iseq` as the 'multiplication' operator. `iseq` is a binary operator that returns 1 if both integer operands are equal and neither of them is 0, and zero otherwise. We denote with $\otimes$ a matrix multiplication using the logical OR and `iseq` semiring. In the following section, we show how $\otimes$ finds bindings of SPARQL query variables in an RDF graph.

Let *RDF selection matrix* be a diagonal matrix with a predicate value on diagonal entries with the indices of rows/columns to be selected. Multiplying an RDF matrix with an RDF selection matrix using the logical OR and `iseq` semiring enables selecting rows from the graph (i.e., nodes) and columns within these rows. The example below demonstrates selecting columns with value b from the second row of matrix $\mathbf{M}$:

$$\underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{\mathbf{S}*b} \otimes \underbrace{\begin{pmatrix} a & b & c \\ a & b & c \\ a & b & c \end{pmatrix}}_{\mathbf{M}} = \underbrace{\begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}}_{C}$$

Matrix $\mathbf{C}$ has a single one at $(2, 2)$, which means that the second row of $\mathbf{M}$ has one cell with the value b in the second column (i.e., $\mathbf{M}(2, 2) = b$).

2.3 SPARQL Query Translation

A *binding matrix* denoted by $\mathbf{M}_{v_1 v_2} : \mathbb{Z}_2^{n \times n}$ is a sparse binary matrix that stores the bindings of SPARQL query edge variables v_1 and v_2. A value of one at index (i, j) in $\mathbf{M}_{v_1 v_2}$ means that i is a binding for variable v_1 and j is a binding for variable v_2. The result set of a SPARQL query can be produced if the binding matrices of all the variables

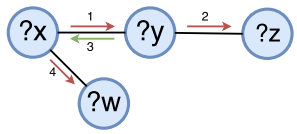

Figure 4: Query graph traversal.

in the query are available. Below we show how to compute the binding matrices for a SPARQL query. We assume the query does not have literals, and does not have cycles for ease of explanation.

A simple single edge SPARQL query, such as:

```
SELECT ?x ?y WHERE {?x <p> ?y .}
```

can be translated to the following semiring matrix multiplication ($\mathbf{S}$ is an RDF selection matrix that captures predicate p):

$$\mathbf{M}_{xy} = \mathbf{S} \otimes \mathbf{A}$$

The RDF selection operation above selects rows that have a p value. In other words, it selects the node pairs (i, j) such that i has an outgoing edge to j with label p, which constitute the valid bindings of variables x and y, respectively.

In a general SPARQL query, each edge is translated to an RDF selection operation. The bindings of one variable are used to find the bindings of the next connected variable. Given a binding matrix of variables x and y, $\mathbf{M}_{xy}$, the bindings of variable y can be converted to an RDF selection matrix by the following operation: $\mathbf{S}_y = \texttt{diag}(\texttt{any}(\mathbf{M}'_{xy}))$, which reduces the columns of $\mathbf{M}_{xy}$ and places the resulting vector from the reduction on the diagonal of an empty matrix. Suppose the query had an edge involving y and z with predicate p_{yz}. The binding matrix $\mathbf{M}_{yz}$ is computed as:

$$\mathbf{M}_{yz} = \mathbf{S}_y * p_{yz} \otimes \mathbf{A}$$

The bindings of y and z in $\mathbf{M}_{yz}$ capture all the edges processed so far; the edge involving (x, y) and the edge involving (y, z). However, some bindings of y in $\mathbf{M}_{xy}$ might have been invalidated by the edge involving (y, z). To accommodate this, the binding matrix $\mathbf{M}_{xy}$ must be updated to select the bindings of y that appear in both binding matrices. This can be done by a column selection operation on $\mathbf{M}_{xy}$, which translates to the following matrix multiplication:

$$\mathbf{M}_{xy} = \mathbf{M}_{xy} \times \texttt{diag}(\texttt{any}(\mathbf{M}_{yz}))$$

MAGiQ translates a query as follows. The undirected version of the query graph is traversed in a depth-first fashion to produce a closed walk such that edges connecting non-leaf nodes appear twice; once when traversing down tree, and once when backtracking. The walk determines the order of the selection operations to be performed on the RDF graph matrix $\mathbf{A}$ to produce a binding matrix for each edge in the query. Edges in the walk have two types: forward edges and backward edges. Forward edges are translated to RDF selection operations that produce the binding matrix for the variables of the query edge. Backward edges are translated to selection operations that filter out invalid variable bindings. Figure 4 demonstrates the traversal done by MAGiQ to produce the matrix algebra program of the example query in Figure 3.

Multiplications over semirings are not yet available in most mature matrix algebra packages, such as MATLAB. However, they are part of the GraphBLAS standard [1], and GraphBLAS conformant implementations such as SuiteSparse:GraphBLAS [4] support them. MAGiQ translates query graphs to matrix multiplications using the standard selection operation by creating a matrix per predicate for the RDF graph. Then each RDF selection operation is replaced with a standard selection operation using the corresponding predicate matrix of the RDF graph.

Table 1: Datasets statistics in millions (M)

Dataset	Triples (M)	#S (M)	#O (M)	#P
LUBM-10240	1,366.71	222.21	165.29	18
YAGO2	284.30	10.12	52.34	98
WatDiv	109.23	5.21	17.93	85

Table 2: LUBM-10240 loading times (minutes).

			MAGiQ	
RDF-3X	**TripleBit**	**Virtuoso**	**SuiteSparse**	**Matlab**
429	101	237	19	**16**

3. EXPERIMENTAL EVALUATION

We show in this section a comparison between our prototype implementation of MAGiQ with multiple matrix algebra backends against state-of-the-art single machine RDF engines. We use the LUBM-10240 dataset with 1.3 billion triples (see Table 1) and its four complex queries [7] L1, L2, L3, and L7. All experiments were executed on a Linux machine with 512GB RAM and Intel Xeon E5-2620 CPU equipped with an NVIDIA Tesla P100 GPU.

Competitors. We compare three implementations of MAGiQ with different backend engines (SuiteSparse:GraphBLAS, MATLAB-CPU, and MATLAB-GPU) against RDF-3X [15], TripleBit [17], and Virtuoso [11]. MAGiQ (SuiteSparse) uses SuiteSparse:GraphBLAS implementation of the GraphBLAS standard and runs on a single CPU thread. MAGiQ (MATLAB-CPU) and MAGiQ (MATLAB-GPU) use MATLAB and run on multiple CPU threads and a single GPU, respectively. RDF-3X is a popular relational RDF engine that uses exhaustive indices to accelerate its join-based query processor. TripleBit uses compact sorted indices and performs merge-joins for query evaluation. Virtuoso is an enterprise grade solution built on top of a hybrid row/column-oriented DBMS. For systems that store indices on disk (RDF-3X and TripleBit), we mounted their indices on memory to make sure all compared systems do not interact with disk while solving queries.

Loading time. Table 2 shows the loading times for the compared systems. MAGiQ takes significantly less time than all other systems; at least 5x faster compared to TripleBit and at most 20x faster compared to RDF-3X. This is because MAGiQ does not build indices, and the loading time is dominated by the time to read the graph from disk.

Query execution time. Table 3 shows the runtimes for all compared systems. MAGiQ with the MATLAB-GPU backend outperforms state-of-the-art specialized engines with a large margin; at least an order of magnitude faster. Even with the MATLAB-CPU backend, MAGiQ also outperforms other systems for most queries. Note that TripleBit did not finish solving L1 within 1 hour, and thus was terminated. The runtimes in Table 3 demonstrate how MAGiQ effortlessly unlocks the power of modern hardware architectures for solving SPAQRL queries through the highly optimized matrix algebra backends.

4. DEMONSTRATION OVERVIEW

Our demonstration illustrates three aspects of MAGiQ: query translation to matrix algebra programs, effortless portability to different hardware architectures, and a direct comparison with the state-of-the-art single machine RDF

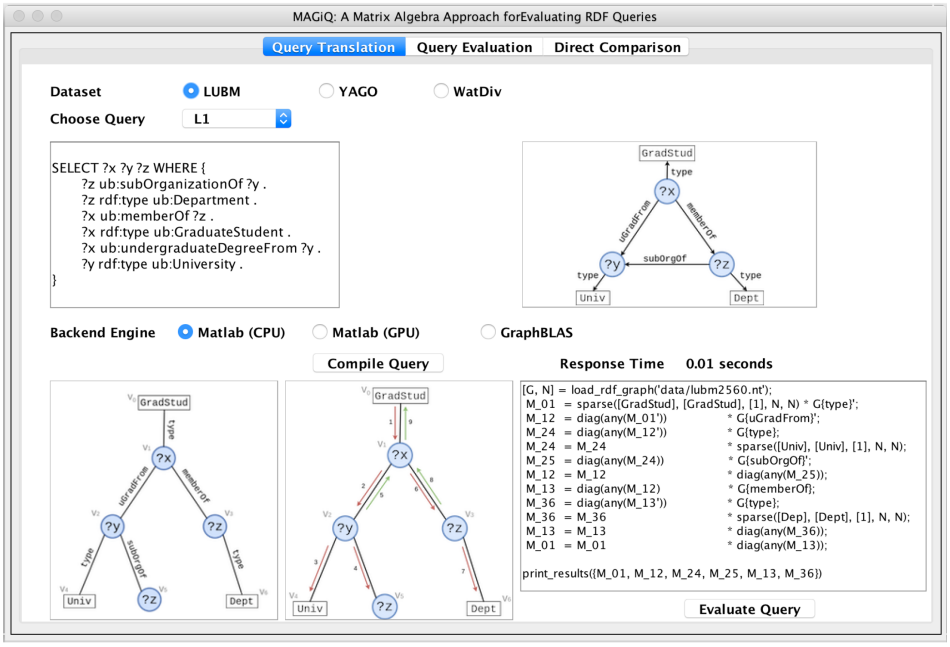

Figure 5: MAGiQ Graphical Interface.

Table 3: Runtimes for LUBM-10240 queries (seconds).

	L1	L2	L3	L7
RDF-3X	1074.6	116.8	1043.6	144.0
TripleBit	N/A	14.2	26.3	65.1
Virtuoso	37.0	86.2	15.6	323.2
MAGiQ (SuiteSparse)	132.4	30.0	85.9	111.9
MAGiQ (Matlab-CPU)	29.5	19.1	6.5	47.2
MAGiQ (Matlab-GPU)	**2.5**	**1.6**	**1.1**	**3.8**

engines. The audience will interact with MAGiQ using our graphical interface shown in Figure 5.

Query translation. The audience will select one of three datasets, and type a SPARQL query. Once a dataset and a query are selected, our graphical interface visualizes the query graph and its traversal, and displays the the resulting matrix algebra program.

Portability. The audience will select one of the currently supported backend engines (i.e., matrix algebra package) to execute the query. The supported backends are: MATLAB and SuiteSparse:GraphBLAS [4]. The MATLAB backend can be instructed to use CPU or GPU.

Comparison. The audience will be able to instruct MAGiQ to run the input query and see a comparison between the response time of MAGiQ and three existing single machine engines; RDF-3X [15], TripleBit [17], and Virtuoso [11].

Datasets. We will use three large-scale real and synthetic datasets: (1) synthetic LUBM-10240 [2] dataset with 1.3 billion triples, (2) real YAGO2 [6] dataset with 284 million triples, and (3) synthetic WatDiv [5] dataset with 109 million triples. Table 1 shows the statistics of each dataset.

5. CONCLUSION

This demo explores sparse matrix algebra as a design paradigm for evaluating SPARQL queries over large RDF datasets. We show that a first attempt in following this paradigm results in an engine that is faster than state-of-the-art engines. We believe that the sparse matrix algebra design paradigm can result in RDF engines with unprecedented portability, scalability, and efficiency.

6. REFERENCES

[1] GraphBLAS standard. `graphblas.org`.
[2] LUBM. `swat.cse.lehigh.edu/projects/lubm`.
[3] RDF Primer. `https://www.w3.org/TR/rdf11-primer/`.
[4] SuiteSparse. `faculty.cse.tamu.edu/davis/suitesparse.html`.
[5] WatDiv. `db.uwaterloo.ca/watdiv`.
[6] YAGO2. `yago-knowledge.org`.
[7] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis. A survey and experimental comparison of distributed sparql engines for very large rdf data. *PVLDB*, 10(13):2049–2060, 2017.
[8] I. Abdelaziz, R. Harbi, S. Salihoglu, and P. Kalnis. Combining vertex-centric graph processing with sparql for large-scale rdf data analytics. *IEEE TPDS*, 28(12):3374–3388, 2017.
[9] I. Abdelaziz, E. Mansour, M. Ouzzani, A. Aboulnaga, and P. Kalnis. Lusail: a system for querying linked data at scale. *PVLDB*, 11(4):485–498, 2017.
[10] A. Buluç and J. R. Gilbert. The combinatorial blas: Design, implementation, and applications. *IJHPCA*, 25(4):496–509, 2011.
[11] O. Erling. Virtuoso, a hybrid rdbms/graph column store. *IEEE Data Eng. Bull.*, 35(1):3–8, 2012.
[12] S. Gurajada, S. Seufert, I. Miliaraki, and M. Theobald. Triad: a distributed shared-nothing rdf engine based on asynchronous message passing. In *ACM SIGMOD*, pages 289–300, 2014.
[13] R. Harbi, I. Abdelaziz, P. Kalnis, N. Mamoulis, Y. Ebrahim, and M. Sahli. Accelerating SPARQL queries by exploiting hash-based locality and adaptive partitioning. *VLDBJ*, 25(3):355–380, 2016.
[14] J. Kepner and J. Gilbert. *Graph algorithms in the language of linear algebra*. SIAM, 2011.
[15] T. Neumann and G. Weikum. RDF-3X: a RISC-style engine for RDF. *PVLDB*, 1(1):647–659, 2008.
[16] M. T. Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.
[17] P. Yuan, P. Liu, B. Wu, H. Jin, W. Zhang, and L. Liu. Triplebit: A fast and compact system for large scale rdf data. *PVLDB*, 6(7):517–528, 2013.
[18] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: Answering sparql queries via subgraph matching. *PVLDB*, 4(8):482–493, 2011.