Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs

Fuad Jamour
KAUST
fuad.jamour@kaust.edu.sa

Ibrahim Abdelaziz
IBM Research
ibrahim.abdelaziz1@ibm.com

Yuanzhao Chen
KAUST
yuanzhao.chen@kaust.edu.sa

Panos Kalnis
KAUST
panos.kalnis@kaust.edu.sa

Abstract

Existing query engines for RDF graphs follow one of two design paradigms: relational or graph-based. We explore sparse matrix algebra as a third paradigm and propose MAGiQ: a framework for implementing SPARQL query engines that are portable on various hardware architectures, scalable over thousands of compute nodes, and efficient for very large RDF datasets. MAGiQ represents the RDF graph as a sparse matrix and defines a domain-specific language of algebraic operations. SPARQL queries are translated into matrix algebra programs that are oblivious to the underlying computing infrastructure. Existing matrix algebra libraries, optimized for each particular architecture, are called to execute the program and handle the performance issues. We present three case studies of matrix algebra back-end libraries: SuiteSparse, Matlab, and CombBLAS; we demonstrate how MAGiQ can effortlessly be ported on a variety of architectures such as Intel CPUs, NVIDIA GPUs, and Cray XC40 supercomputers. Our experiments on large-scale real and synthetic datasets show that MAGiQ performs comparably to or better than existing specialized SPARQL query engines for data-intensive queries, scales to very large computing infrastructures, and handles datasets with up to 512 billion triples.

Keywords Graph Query Engines, RDF, Matrix Algebra

ACM Reference Format:
Fuad Jamour, Ibrahim Abdelaziz, Yuanzhao Chen, and Panos Kalnis. 2019. Matrix Algebra Framework for Portable, Scalable and Efficient Query Engines for RDF Graphs. In *Fourteenth EuroSys Conference 2019 (EuroSys '19), March 25–28, 2019, Dresden, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3302424.3303962

EuroSys '19, March 25–28, 2019, Dresden, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6281-8/19/03…$15.00
https://doi.org/10.1145/3302424.3303962

1 Introduction

RDF [10] data are collections of triples of the form ⟨`subject`, *predicate*, `object`⟩, where predicates describe relationships between subjects and objects, e.g., ⟨`apple`, *isA*, `fruit`⟩. RDF data can be viewed as directed, edge-labelled graphs where each triple corresponds to an edge. The RDF model is popular in many application domains such as the semantic web, bioinformatics, and knowledge graphs [13, 38, 46].

SPARQL [7] is the de-facto query language and there exist many systems that support SPARQL queries over large RDF datasets [13, 46]. Two design paradigms are dominant: the relational paradigm, which builds exhaustive indices and utilizes relational operators (e.g., joins) to solve SPARQL queries [34, 35, 45]; and the graph-based paradigm, which represents RDF data in its native graph form and uses graph traversal for query evaluation [14, 55, 56]. Regardless of the paradigm, all existing solutions are tightly coupled to a particular hardware architecture, typically CPUs [34, 45, 56]. However, modern computing query engines are equipped with diverse hardware configurations; for example, it is common to find GPUs. Unfortunately, few existing SPARQL query engines utilize GPUs [26, 51]. Adapting existing query engines to utilize GPUs takes substantial engineering effort even though the underlying ideas for query planning and execution are well established in the literature.

The High Performance Computing (HPC) community has been dealing with a similar problem for decades. For each novel NUMA configuration (e.g., non-volatile RAM), many-core accelerator (e.g., Intel Phi), GPU system (e.g., NVIDIA NVLink), or proprietary supercomputer network (e.g., Cray Aries), they do not reimplement their partial differential equation solvers or heat transfer simulations from scratch. Instead, they rely on basic algebraic operations libraries, such as BLAS [1]. These libraries are optimized for each new architecture and they alleviate the burden of difficult tasks such as data partitioning, cache locality, parallel execution, synchronization, communication, and load balancing.

Interestingly, since RDF data correspond to graphs, they can be represented as adjacency matrices. The question that arises is whether we can use matrix algebra, and hence rely on existing libraries, to perform useful processing on graphs.

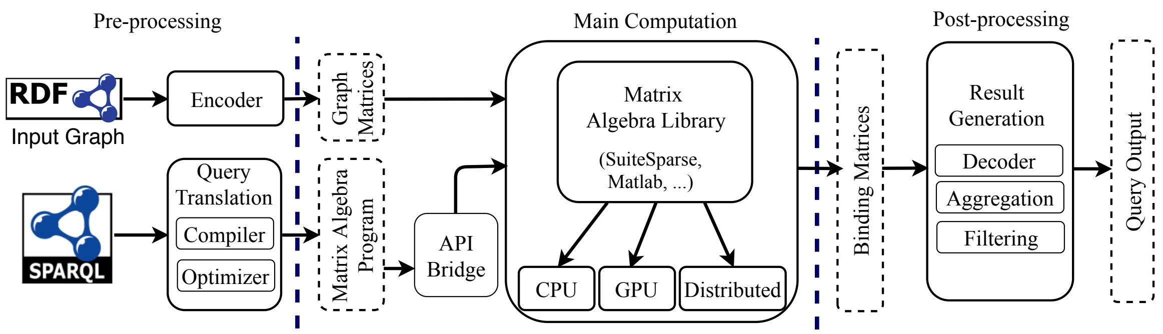


Figure 1. Overview of MAGiQ. The pre-processing, main computation, and post-processing phases are shown.

It turns out there are several research efforts towards this direction [41], culminating to GraphBLAS [3] that defines a standard set of sparse matrix algebra primitives for solving graph problems. Several implementations of the standard are available such as SuiteSparse:GraphBLAS [8]. Matrix algebra libraries have been used to implement several graph algorithms. However, these algorithms either have well-defined algebraic formulations as in the case of PageRank [17, 31], betweenness centrality [23, 49], and triangle counting [28]; or are simple graph traversals such as BFS [19]. To the best of our knowledge, there is no implementation of something as complex as a SPARQL query engine.

We propose MAGiQ[1]: a matrix algebra framework for implementing SPARQL query engines. MAGiQ defines a domain-specific language of matrix operations. The language consists of the usual matrix algebraic operations, with the addition of a custom matrix multiplication operation defined on a modified semiring that replaces the usual arithmetic addition and multiplication with logical OR and equality operator, respectively. MAGiQ stores the RDF graph as a sparse integer matrix, and translates SPARQL queries into concise matrix algebra programs that operate on the matrix representation. These programs are oblivious to the underlying computing infrastructure and are executed by existing back-end matrix libraries, optimized for each particular hardware architecture. Developers only need to implement an API bridge between the domain-specific language of MAGiQ and the back-end library; the concept is similar to the ODBC/JDBC drivers for databases. We present three case studies using SuiteSparse:GraphBLAS [8] for CPUs, MATLAB for CPUs and GPUs, and CombBLAS [23] for distributed-memory systems. We describe the corresponding API bridges and demonstrate that the required effort to support additional libraries is negligible.

In contrast to existing specialized SPARQL query engines, the sparse matrix algebra paradigm eliminates the need for building exhaustive indices; consequently, MAGiQ loads data 3x to 28x faster[2]. The memory footprint is also reduced, enabling us to load an RDF dataset with 4.3 billion triples on a single machine. In comparison, the largest dataset used in several state-of-the-art engines [34, 48, 51] contains 1.3 billion triples. Moreover, MAGiQ inherits the good scalability of the underlying libraries, both in terms of data size and number of compute nodes. We managed to process a dataset with 512 billion triples and utilized up to 2,048 distributed-memory compute nodes on a Cray XC40 supercomputer. Interestingly, the improved portability and scalability of MAGiQ do not pose an adverse effect to performance, especially for data-intensive queries (i.e., queries whose evaluation generates large intermediate results) that challenge most query engines. Using the same hardware configuration to solve the data-intensive queries of the established LUBM [5] benchmark, MAGiQ is in the worst case 2.2x slower, but it can be up to 147.2x faster than existing query engines.

This paper is organized as follows. Section 2 gives a high level overview of MAGiQ, while Sections 3 and 4 explain how SPARQL queries are translated into matrix algebra programs. Section 5 describes case studies with three back-end libraries. Our experimental evaluation is presented in Section 7, and Section 9 concludes the paper.

2 Overview of MAGiQ

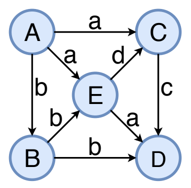
Figure 1 shows the architecture of MAGiQ. Similarly to HPC workflows, there are three phases: pre-processing, main computation, and post-processing. Pre- and post-processing phases are lightweight, while most of the query evaluation time is spent in the main computation phase.

2.1 RDF Graph Representation

MAGiQ stores the RDF graph as sparse square matrix $\mathbf{A} : \mathbb{Z}^{n \times n}$, where n is the number of nodes in the graph (i.e., the number of unique subjects and objects). A non-zero entry $\mathbf{A}(i, j) = p_{ij}$ denotes that subject i is connected to object j with predicate p_{ij}. $nnz(\mathbf{A})$ is the number of non-zero entries

[1]A demo of MAGiQ was presented in VLDB '18 [37].

[2]The trade-off is slower execution for selective queries whose evaluation benefits from indices and generates small intermediate results.



$$\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & & b & a & & a \\ B & & & & b & b \\ C & & & & c & \\ D & & & & & \\ E & & & d & a & \end{array}$$

Figure 2. Example RDF graph (left), and corresponding RDF sparse matrix (right).

in **A**, i.e., the number of edges in the graph. Row $\mathbf{A}(i,:)$ stores the predicates of the outgoing edges of node i. Figure 2 shows an example RDF graph with 5 nodes $A \dots E$ and 4 unique predicates $a \dots d$. There are 8 triples resulting in 8 non-zero entries in **A**.

The specific data structure used to store **A** is out of the scope of this paper; the choice is left to the matrix algebra back-end. However, it is worth noting that space efficient data structures for storing sparse matrices are available, such as the compressed sparse column (CSC) [30], the doubly compressed sparse column (DCSC) [24], and the coordinate list (COO) format. These data structures have linear space complexity in the number of edges of the input graph, which enables us to support very large RDF graphs.

It is possible for RDF datasets to contain multi-edges, i.e., several edges connecting the same pair of nodes with different predicates. Multi-edges are seamlessly supported by the COO format. CSC and DCSC, on the other hand, do not support multi-edges. A simple workaround to support multi-edges is to store multiple binary matrices, one matrix per predicate (i.e., one CSC or DCSC structure per predicate), to indicate the existence of an edge with a certain predicate. This resembles a predicate-based index where the key is the predicate and the value is the predicate matrix.

The COO format also supports efficient updates. Adding/removing triples to/from the RDF graph corresponds to adding/removing an entry to/from the COO list. Our Matlab implementation uses the COO format.

2.2 Query Evaluation Workflow

Similarly to most SPARQL query engines [13, 14, 18, 35, 45], MAGiQ starts by loading the input RDF graph, encoding its strings into numerical IDs, and building a bi-directional dictionary. The encoded graph is represented as a sparse square integer matrix. Then, MAGiQ translates SPARQL queries into matrix algebra programs, and the optimizer utilizes matrix algebra properties to reorder the operations to generate more efficient programs. An existing sparse matrix algebra back-end library is used to evaluate the program over different hardware architectures. For example, one can use Matlab for CPU and GPU, SuiteSparse:GraphBLAS [8] or MKL [4] for CPU, cuSPARSE [6] for GPUs, or CombBLAS [23] for distributed-memory systems. MAGiQ's API bridge maps the intermediate matrix algebra program to the corresponding

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

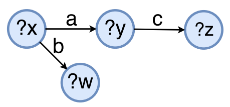


Figure 3. Example SPARQL query (left), and its graph representation (right); each triple pattern in the query maps to an edge in the query graph. The answer to this query when evaluated against the RDF graph in Figure 2 is $(x, y, z, w) \in \{(A, C, D, B)\}$.

back-end syntax. The back-end gets as inputs both the graph matrix and the matrix algebra program. It performs a sequence of matrix operations that generate a set of binding matrices containing the matches of the query variables; these are passed to the result generation phase that constructs the final query answer.

We focus on conjunctive SPARQL queries (a.k.a. basic graph patterns) because such queries are building blocks of most other SPARQL queries [7], similarly to most SPARQL query engines [14, 18, 34, 35, 45, 48, 55]. Basic conjunctive SPARQL queries take the form (syntactic sugar aside):

SELECT $?v_1$ ... $?v_N$ WHERE tp_1 . ... tp_M .

Such queries ask for bindings of variables $v_1 \dots v_N$ from the input RDF graph; the returned bindings should satisfy the connectivity restrictions defined in the triple patterns $tp_1 \dots tp_M$. MAGiQ supports triple patterns with variable/constant subjects and objects, and constant predicates[3]. These queries can be represented as graphs (see Figure 3); we refer to query triple patterns as query edges hereafter.

Example: Consider the example SPARQL query in Figure 3. This query is translated into the following matrix algebra program, where query edges are processed in the following order: ⟨?x, a, ?y⟩, ⟨?y, c, ?z⟩, and ⟨?x, b, ?w⟩:

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

The $\otimes$ symbol denotes matrix multiplication over a semiring (explained in Section 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 **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 $\mathbf{M}_{xy}$ 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.

[3]SPARQL allows variable predicates in triple patterns; however, query predicates are typically constants [21, 48]. SPARQL specifications also describe other query constructs such as property path expressions and optional statements that we do not cover in this paper. We refer interested readers to studies of real SPARQL query logs for more information on the significance of the different SPARQL constructs [21, 33].

3 Matrix Algebra Constructs

This section introduces the matrix algebra constructs that serve as the main building blocks in our domain-specific language. These constructs are oblivious to any specific implementation; the underlying data structures and algorithms used to realize them vary across different back-ends.

3.1 Selection Operation in RDF Matrices

Row/column selection. A *selection matrix* is a diagonal matrix with 1s on diagonal entries at 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. Placing the selection matrix on the left side of the multiplication results in row selection, while placing it on the right side results in column selection. We refer to the multiplication of a matrix **A** with a selection matrix **S** as *selection operation*.

The following example selects a row from **A**. Let selection matrix **S** have a single 1 at index $(2, 2)$. The operation results in matrix **M**, where only the second row of **A** is present. The same operation can extract multiple rows by placing more 1s on the diagonal of the selection matrix.

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

Semirings. A semiring is a set with two binary operators, "addition" and "multiplication" [41]. A matrix algebra can be defined over semirings other than the standard arithmetic addition and multiplication. We use a semiring with the set of integers, `LOR` as the "addition" operator, and `iseq` as the "multiplication" operator. `LOR` is logical OR operator and `iseq` is a binary operator that returns 1 if both integer operands are equal and neither of them is 0, or zero otherwise. We use $\otimes$ to denote matrix multiplication using the `LOR` and `iseq` semiring[4].

Row/column selection with predicates. Let *predicate selection matrix* be a diagonal matrix with a predicate value on diagonal entries corresponding to rows/columns to be selected. Applying the $\otimes$ operation between the RDF matrix and a predicate selection matrix selects rows from the graph (i.e., nodes) and columns within these rows with matching predicate value; we call this *predicate selection operation*. The example below demonstrates the selection of those cells of **A** that belong to the 1st or 3rd row and contain value b.

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

Predicate selection matrix $\mathbf{S}^b$ is produced by multiplying **S** with scalar b using the usual scalar multiplication operation $*$. Matrix **M** contains two 1s at $(1, 2)$ and $(3, 2)$, corresponding to cells $\mathbf{A}(1, 2)$ and $\mathbf{A}(3, 2)$ with value b.

Binding matrices. We refer to the matrix that results from a selection operation as *binding matrix* denoted by $\mathbf{M}_{v_1 v_2} : \mathbb{Z}_2^{n\times n}$. This is a sparse binary matrix that stores the bindings of SPARQL query edge variables v_1 and v_2. A value of 1 at index (i, j) in $\mathbf{M}_{v_1 v_2}$ means that i and j are bindings for variables v_1 and v_2, respectively. Each edge in a query needs a binding matrix to store bindings of its variables.

3.2 Matrix Algebra Building Blocks

We describe below the seven operations defined in the domain-specific language of MAGiQ; Table 1 contains a summary. The API bridge needs only to provide mappings from these operations to the corresponding function calls of each back-end matrix algebra library.

Scalar multiplication. Denoted by $*$, this operation receives selection matrix $\mathbf{S} : \mathbb{Z}_2^{n\times n}$ and integer p, and produces matrix $\mathbf{S}^p : \mathbb{Z}^{n\times n}$ with the 1s of **S** multiplied by p.

Matrix multiplication with `LOR.iseq` semiring. Denoted by $\otimes$, this operation receives predicate selection matrix $\mathbf{S}^p : \mathbb{Z}^{n\times n}$ and RDF matrix $\mathbf{A} : \mathbb{Z}^{n\times n}$, and produces a binding matrix $\mathbf{M} : \mathbb{Z}_2^{n\times n}$.

Standard matrix multiplication. Denoted by $\times$, this operation receives selection matrix $\mathbf{S} : \mathbb{Z}_2^{n\times n}$ and binding matrix $\mathbf{M} : \mathbb{Z}_2^{n\times n}$, and produces an updated binding matrix $\mathbf{M} : \mathbb{Z}_2^{n\times n}$.

Reduction with `LOR`. Denoted by `any` (inspired by Matlab's `any` function), this operation receives binding matrix $\mathbf{M} : \mathbb{Z}_2^{n\times n}$ and produces column vector $\mathbf{v} : \mathbb{Z}_2^{n\times 1}$. Each element $\mathbf{v}(i)$ is the result of applying the `LOR` operator on the elements of row i in **M**.

Selection matrix construction. Denoted by `diag` (inspired by Matlab's `diag` function), this operation receives binding vector $\mathbf{v} : \mathbb{Z}_2^{n\times 1}$ and produces selection matrix $\mathbf{S} : \mathbb{Z}_2^{n\times n}$ with the elements of **v** on the main diagonal.

Vector constructor. Denoted by `one`, this operation receives query constant integer i and produces binding vector $\mathbf{v} : \mathbb{Z}_2^{n\times 1}$ with a single 1 at index i.

Matrix transposition. Denoted by a single quote ($'$), this operation receives matrix $\mathbf{M} : \mathbb{Z}^{n\times n}$ and produces the transposed matrix $\mathbf{M}' : \mathbb{Z}^{n\times n}$.

[4]Formally, the operators in a semiring should contain identity elements; this is not satisfied by `iseq`. However, in the context of GraphBLAS and its surrounding literature, it is common to refer to semiring-like constructs as semirings.

Table 1. Summary of MAGiQ's matrix algebra building blocks.

Operation	Semantic
$\mathbf{S} * p$	Multiply matrix **S** by integer p
$\mathbf{S} * p \otimes \mathbf{A}$	Select nodes specified in **S** with predicate p from RDF graph **A**
$\mathbf{M} \times \mathbf{S}$	Select nodes specified in **S** from binding matrix **M**
$\mathsf{any}(\mathbf{M})$	Extract query variable bindings from binding matrix **M**
$\mathtt{diag}(\mathbf{v})$	Produce a selection matrix from query variable bindings in **v**
$\mathsf{one}(i)$	Produce a binding vector with a single constant i
$\mathbf{M}'$	Transpose matrix **M**

4 SPARQL to Matrix Algebra

In this section we explain how a conjunctive SPARQL query is translated in MAGiQ; the query graph is translated into a matrix algebra program that uses the operations described in Section 3. The outcome of the matrix algebra program is a collection of binding matrices that hold the bindings of each pair of variables specified in a query edge. The binding matrices are finally used in the result generation phase to produce the query results (Section 6).

4.1 Query Translation

Single edge query translation. We show below how to compute the binding matrices for a simple SPARQL query. Consider the following query to be evaluated over the RDF graph in Figure 2.

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

This single edge query asks for all pairs of nodes x and y with an a-labelled edge from x to y. Such node pairs constitute the valid bindings of variables x and y, respectively. The query is translated to a predicate selection operation. Since no prior variable bindings are available, we must consider the entire domain. Therefore, the predicate selection matrix is the identity matrix **I** multiplied by predicate a (i.e., $\mathbf{S} = \mathbf{I} * a$). The variable bindings are stored in binding matrix $\mathbf{M}_{xy}$:

$$\mathbf{M}_{xy} = \mathbf{I} * a \otimes \mathbf{A}$$

The operation is calculated as follows:

$$\mathbf{M}_{xy} = \underbrace{\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & a & & & & \\ B & & a & & & \\ C & & & a & & \\ D & & & & a & \\ E & & & & & a \end{array}}_{\mathbf{I}*a} \otimes \underbrace{\begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & & b & a & & a \\ B & & & & b & b \\ C & & & & c & \\ D & & & & & \\ E & & & d & a & \end{array}}_{\mathbf{A}}$$

and results in:

$$\mathbf{M}_{xy} = \begin{array}{c|ccccc} & A & B & C & D & E \\ \hline A & & & 1 & & 1 \\ B & & & & & \\ C & & & & & \\ D & & & & & \\ E & & & & 1 & \end{array}$$

The bindings of x and y are: $(x, y) \in \{(A, C), (A, E), (E, D)\}$.

Graph query translation. For a graph query, each edge is translated to a predicate selection operation. The bindings of a variable constrain the bindings of the next connected variable. Given a binding matrix $\mathbf{M}_{v_1 v_2}$, the bindings of variable v_2 can be converted to a selection matrix as follows:

$$\mathbf{S}_{v_2} = \mathtt{diag}(\mathsf{any}(\mathbf{M}'_{v_1 v_2}))$$

where the any operation returns a vector with all bindings of v_2 and the `diag` operation places the resulting vector on the diagonal of an empty matrix. Suppose the query had another edge involving v_2 and v_3 with predicate $p_{v_2 v_3}$. The binding matrix $\mathbf{M}_{v_2 v_3}$ is computed as follows:

$$\mathbf{M}_{v_2 v_3} = \mathbf{S}_{v_2} * p_{v_2 v_3} \otimes \mathbf{A}$$

The bindings in $\mathbf{M}_{v_2 v_3}$ satisfy the constraints of all edges processed so far, i.e., (v_1, v_2) and (v_2, v_3). However, some bindings of v_2 in $\mathbf{M}_{v_1 v_2}$ may have been invalidated by the constraint of edge (v_2, v_3). To accommodate this, $\mathbf{M}_{v_1 v_2}$ must be updated to include only the bindings of v_2 that appear in both binding matrices. This is done as follows (refer to column selection in Section 3.1):

$$\mathbf{M}_{v_1 v_2} = \mathbf{M}_{v_1 v_2} \times \mathtt{diag}(\mathsf{any}(\mathbf{M}_{v_2 v_3}))$$

Note that if a binding of v_1 is left without matching bindings of v_2, the operation above eliminates such bindings of v_1.

Algorithm Query-Translation shows how MAGiQ translates acyclic graph queries (i.e., tree queries) without constants to matrix algebra programs; we describe the required modifications for queries with cycles and constants in Section 4.2. First, the undirected version of the query graph is traversed in a depth-first fashion to produce a closed walk (Line 3) such that edges connecting non-leaf nodes appear twice: once when traversing down the tree, and once when backtracking. In DFS-walk, an edge is labelled *forward* edge when it is discovered, and edges encountered while backtracking are labelled *back* edges. Then, the extracted walk (*qwalk* in Query-Translation) guides the program generation through the loop in Lines 4-21.

The first edge in *qwalk* results in a $\otimes$ operation, where the first matrix is the identity matrix multiplied by the predicate of the respective edge, and the second matrix is either the RDF matrix **A** or its transpose $\mathbf{A}'$ (Lines 10–12). $\mathbf{A}'$ is used when the direction of e in *qwalk* does not match that in the directed query graph, in which case the row indices in the

```
Algorithm: QUERY-TRANSLATION
Input: Query graph $\mathbf{q}(V_q, E_q)$
Output: Matrix algebra program: statements $s_0 \ldots s_k$ each with 4
        components $s_i.\mathbf{M}_{out}$, $s_i.\mathbf{M}_{in_1}$, $s_i.\mathbf{M}_{in_2}$, and $s_i.op$
1  Let $S = \phi$
2  Let $\mathbf{g}$ be the undirected version of $\mathbf{q}$
3  $qwalk$ = DFS-WALK($\mathbf{g}$) // Forward DFS and backtracking walk
4  for $e \in qwalk$ do
5      Let $pe$ be the edge before $e$ in $qwalk$
6      Let $p$ be the predicate of edge $e$
7      $(v_1, v_2, etype) = (e.v_1, e.v_2, e.type)$
8      $(w_1, w_2) = (pe.v_1, pe.v_2)$
9      if $pe.type$ == 'back' then $(w_1, w_2) = (pe.v_2, pe.v_1)$
10     if $e$ is the first edge in $qwalk$ then
11         $(\mathbf{M}_{out}, \mathbf{M}_{in_1}, \mathbf{M}_{in_2}, op) = (\mathbf{M}_{v_1 v_2}, \mathbf{I} * p, \mathbf{A}, \otimes)$
12         if $e \notin E_q$ then $\mathbf{M}_{in_2} = \mathbf{A}'$
13     else if $etype$ == 'forward' then
14         $(\mathbf{M}_{out}, \mathbf{M}_{in_1}, \mathbf{M}_{in_2}, op) =$
           $(\mathbf{M}_{v_1 v_2}, \mathtt{diag}(\mathtt{any}(\mathbf{M}_{w_1 w_2})) * p, \mathbf{A}, \otimes)$
15         if $v_1 \neq w_1$ then
16             $\mathbf{M}_{in1} = \mathtt{diag}(\mathtt{any}(\mathbf{M}'_{w_1 w_2})) * p$
17         if $e \notin E_q$ then $\mathbf{M}_{in_2} = \mathbf{A}'$
18     else if $etype$ == 'back' then
19         $(\mathbf{M}_{out}, \mathbf{M}_{in_1}, \mathbf{M}_{in_2}, op) =$
           $(\mathbf{M}_{v_2 v_1}, \mathbf{M}_{v_2 v_1}, \mathtt{diag}(\mathtt{any}(\mathbf{M}_{w_1 w_2})), \times)$
20     $s = (\mathbf{M}_{out}, \mathbf{M}_{in_1}, \mathbf{M}_{in_2}, op)$
21     $S$.append(s)
22 return $S$
```

resulting binding matrix correspond to nodes with incoming edges with predicate p in the RDF graph, or rows in $\mathbf{A}'$. Edges in *qwalk* with type *forward* result in a $\otimes$ operation, where the first operand is a selection matrix with the bindings of the previous variable w_1, if the current edge e and the previous edge pe share the same query variable as a first node (Lines 13–17). Otherwise the bindings of pe's second variable are used in the selection matrix (Line 16). If the direction of e in *qwalk* does not match that of the corresponding edge in the directed query graph, the next bindings are those of nodes with outgoing edges to the previous bindings, or equivalently rows in $\mathbf{A}'$; thus, the row selection is done on $\mathbf{A}'$ (Line 17). Finally, edges with type *back* result in column selection on the binding matrix under consideration to eliminate bindings invalidated by another selection (Lines 18–19).

Example: Consider the query of Figure 3 on the RDF graph of Figure 2. Assume the query graph is traversed starting from node x to generate the following *qwalk* (as shown in Figure 4): (x, y, *forward*), (y, z, *forward*), (y, x, *back*), (x, w, *forward*). The first edge translates (Line 11) to a selection of all rows with predicate a in $\mathbf{A}$: $\mathbf{M}_{xy} = \mathbf{I} * a \otimes \mathbf{A}$. After this step, $\mathbf{M}_{xy}$ contains three non-zeros at (A, C), (A, E), and (E, D). The second edge is a *forward* edge, and the first node in e and pe are not equal (v_1 is y, while w_1 is x), so the selection matrix $\mathbf{M}_{in_1}$ should have the bindings of $w_2 = y$, and thus $\mathbf{M}_{xy}$ has to be transposed before constructing the selection matrix

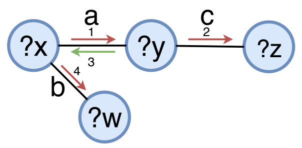


Figure 4. DFS-WALK over the query in Figure 3 starting from node x. Red arrows represent *forward* edges while green arrows are *back* edges.

$\mathbf{M}_{in_1} = \mathtt{diag}(\mathtt{any}(\mathbf{M}'_{xy}))$. The statement for this edge extracts nodes connected to bindings of y with an outgoing edge from y, or equivalently rows in $\mathbf{A}$ (Lines 14 and 16). The resulting statement is: $\mathbf{M}_{yz} = \mathtt{diag}(\mathtt{any}(\mathbf{M}'_{xy})) * c \otimes \mathbf{A}$. After this step, $\mathbf{M}_{yz}$ has one non-zero at (C, D). The third edge is a *back* edge, which updates the bindings in $\mathbf{M}_{xy}$ to eliminate bindings invalidated by the previous selection (computing $\mathbf{M}_{yz}$). $\mathbf{M}_{xy}$ is updated by selecting the columns that are bindings of y in $\mathbf{M}_{yz}$, or $\mathtt{diag}(\mathtt{any}(\mathbf{M}_{yz}))$. The resulting statement is (from Line 19): $\mathbf{M}_{xy} = \mathbf{M}_{xy} \times \mathtt{diag}(\mathtt{any}(\mathbf{M}_{yz}))$. After this step, $\mathbf{M}_{xy}$ has one non-zero at (A, C). Finally the last edge is a *forward* edge where both v_1 and w_1 are equal to x, in which case a row selection is done on $\mathbf{A}$ using bindings of x in the selection matrix (Line 14), resulting in the following statement: $\mathbf{M}_{xw} = \mathtt{diag}(\mathtt{any}(\mathbf{M}_{xy})) * b \otimes \mathbf{A}$. After this step, $\mathbf{M}_{xw}$ has one non-zero at (A, B).

4.2 General Conjunctive SPARQL Queries

So far, we discussed the translation of conjunctive SPARQL queries that are acyclic and without constants. MAGiQ handles queries with cycles and constants as follows.

Cycles. Queries with cycles are converted to acyclic queries and then translated using Algorithm QUERY-TRANSLATION. The conversion to acyclic queries happens as follows. The undirected version of the input query is traversed in a depth-first fashion. When exploring a node v_e, if a neighbour node v_n is discovered and it is not a parent of v_e (i.e., a DFS back edge (v_e, v_n) is encountered), edge (v_e, v_n) is replaced with an edge connecting v_e and v_s, where v_s is a newly introduced *shadow* node of v_n, i.e., newly introduced shadow query variable that represents an existing query variable. The result generation phase (Section 6) is aware of these extra shadow variables and it does not output bindings for them, but it uses their bindings to produce correct bindings for the original query variables.

Example: Figure 5 demonstrates the conversion of a cyclic query (Figure 5a) to an acyclic query (Figure 5c). The process goes as follows (Figure 5b), assuming the traversal starts from node x. In the first two steps, nodes y and z are discovered. Then, when the discovered node x is encountered through back edge (z, x), this edge is removed and new edge (z, w) is introduced with the new shadow node w. Note that the traversal is performed on the undirected version of the input query graph, but the original edge directions and labels are maintained in the resulting acyclic query graph.

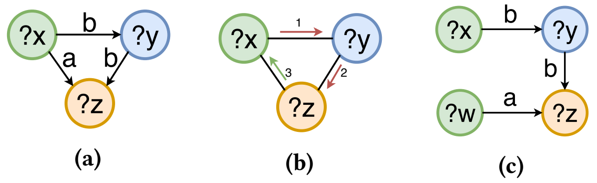


Figure 5. Cyclic query example: (a) Input query. (b) DFS process; red and green denote DFS tree and back edges, respectively. (c) Acyclic query; *w* is a shadow variable of *x*.

Queries with constants. A constant l in an edge implies selecting nodes connected to node l in the RDF graph, or equivalently selecting a specific row or column from $\mathbf{A}$. Constants are handled as follows. If the constant is the first node in the first edge in *qwalk*, the selection matrix $\mathbf{I}*p$ is replaced with $\mathtt{diag}(\mathtt{one}(l)) * p$. Otherwise, $\mathbf{A}$ (or $\mathbf{A}'$) is replaced with $\mathbf{A} \times \mathtt{diag}(\mathtt{one}(l))$ (or $\mathbf{A}' \times \mathtt{diag}(\mathtt{one}(l))$, respectively) when the edge involving the constant is processed.

Queries with variable predicates. Some SPARQL queries contain triple patterns with variable predicates. If each variable predicate's name appears in a single triple pattern in the query, several small changes are needed to support such queries in MAGiQ. The main change is replacing `iseq` in our `LOR.iseq` semiring with an operator that returns the second operand if both operands are non-zeros and returns zero otherwise. In the general case when the same predicate variable appears in multiple triple patterns in the query (i.e., the query requires joins involving predicates), non-trivial changes are required to support those queries in MAGiQ. It is worth noting that queries with variable predicates are not commonly encountered in real SPARQL queries as confirmed in several studies of real SPARQL query logs [21, 33]. We defer supporting variable predicates to our future work.

4.3 Query Optimization

The query evaluation time in MAGiQ depends mainly on the efficiency of the matrix algebra back-end. However, for a given query, different equivalent matrix algebra programs can result in different runtimes. One factor that contributes to the efficiency of a query program, regardless of the back-end, is the number of non-zeros in binding matrices. Some binding matrices may become as dense as the matrix of the input graph. During processing, some may become sparser after a *back* edge removes invalidated bindings. Different orders of processing the query edges can result in programs with varying efficiency.

The selection of the starting node in Algorithm DFS-WALK controls the order of processing query edges. Since query optimization is not the focus of this paper, we follow a simple query planning strategy where we start with the edge that involves a constant, in the hope that such an edge is selective and that the resulting binding matrices remain sparse while evaluating the query. We show in Section 7.1.4 that the query evaluation program produced by this simple strategy often picks either the fastest possible plan, or close to it. We defer more involved optimization approaches to our future work.

MAGiQ uses a simple optimization for cyclic queries to take into account the fact that two (or more) nodes represent the same query variable (i.e., original variable and its shadow variables). Recall from Section 4.1 that *back* edges result in filtering out bindings from already computed binding matrices. We use the same idea to filter out invalid bindings of a query variable that has shadow variables. This optimization reduces the number of non-zeroes in the intermediate binding matrices and leads to faster result generation.

MAGiQ also takes into account the storage format of the sparse matrix in the back-end library. For libraries with column major data structures such as compressed sparse columns, column selection operations are faster and should be used instead of row selection where possible. This can be done by computing the transpose of a binding matrix, instead of the binding matrix itself. Consider row selection operation $\mathbf{M} = \mathbf{S} \otimes \mathbf{A}$; the transpose $\mathbf{M}'$ equals $\mathbf{A}' \otimes \mathbf{S}'$. Since $\mathbf{S}$ is a square diagonal matrix, it is equal to its transpose. Thus, statement $\mathbf{M}' = \mathbf{A}' \otimes \mathbf{S}$ is used where possible when the back-end uses column major format.

5 Main Computation - Implementation

We present three case studies with different back-end matrix algebra libraries: SuiteSparse, MATLAB, and CombBLAS. We map the domain-specific language of MAGiQ to the functions of the back-end library by implementing the API bridge of Figure 1. Additional libraries can be easily supported by implementing the corresponding API bridge.

MAGiQ (SuiteSparse). We use the SuiteSparse [8] implementation of the GraphBLAS [3] standard. GraphBLAS offers `GrB_Matrix` and `GrB_Vector` data types for storing matrices and vectors. RDF graph construction is done with `GrB_Matrix_build`, which constructs a `GrB_Matrix` from $\langle rowIndex, columnIndex, value\rangle$ tuples. Since this library supports the GraphBLAS standard, mapping our operations is straightforward. $\times$ and $\otimes$ map to `GrB_mxm` with a `GxB_PLUS_TIMES` and `GxB_LOR_EQ` semirings, respectively. Scalar multiplication and `diag` map to calls to `GrB_Matrix_setElement` in `GrB_NONBLOCKING` mode, which avoids reconstructing the matrix upon each call. `any` maps to `GrB_Matrix_reduce_Monoid` with `GxB_LOR_BOOL_MONOID` binary operator. Operation `one` maps to `GrB_Vector_setElement`. Transposing a matrix maps to `GrB_transpose`. Since this library uses column major storage format (as of version 1.1.0), columns in a matrix are stored contiguously in memory; consequently, column selection operations are much faster. MAGiQ adjusts the query evaluation programs for this library such that row selection operations are replaced with column selection operations. The RDF matrix and its transpose are stored to avoid the redundant computation of $\mathbf{A}'$.

```
Algorithm: RESULT-GENERATION
Input: Binding matrices $\mathbf{M}^{e_1}_{v_1 v_2} \dots \mathbf{M}^{e_n}_{v_1 v_2}$
$\mathbf{M}^{e_i}_{v_1 v_2}$ is the binding matrix for edge $e_i$ in the undirected query graph
$\mathbf{g}$, which has the bindings of $e_i.v_1$ and $e_i.v_2$
Output: Query results table $R$
1 $R$ = Get-IJ ($\mathbf{M}^{e_1}_{v_1 v_2}$)
2 for $M^{e_i}_{v_1 v_2} \in M^{e_2}_{v_1 v_2} \dots M^{e_n}_{v_1 v_2}$ do
3     $L$ = Get-IJ ($M^{e_i}_{v_1 v_2}$)
4     $R$ = JOIN ($R$, $L$)
5     if $v_2$ is a shadow variable then $R$ = FILTER ($R$)
6 return $R$
```

MAGiQ (Matlab). Matlab, like most mature matrix algebra pacakges, does not support matrix multiplication over semirings. However, the `LOR.iseq` semiring multiplication can be implemented using standard matrix multiplication by decomposing the input RDF graph in multiple sparse binary matrices A_p, called *predicate matrices*. A predicate matrix A_p can be extracted from the RDF matrix $\mathbf{A}$ with the Matlab comparison operator ==, that is $\mathbf{A}_p = (\mathbf{A} == p)$. Consequently, a `LOR.iseq` semiring multiplication $\mathbf{M}_{v_1 v_2} = \mathbf{S} * p \otimes \mathbf{A}$ can be implemented in Matlab as: $\mathbf{M}_{v_1 v_2} = \mathbf{S} \times \mathbf{A}_p$. The sparse matrix-matrix multiplication in Matlab is slow on GPUs (as of version R2017B), so we replace matrix-matrix multiplications with matrix-vector multiplications and compute binding vectors instead of binding matrices. Binding matrices are then computed from the binding vectors and $\mathbf{A}$ using Matlab vectorized operations.

MAGiQ (CombBLAS). This implementation uses CombBLAS [23]: a distributed matrix algebra library that supports matrix multiplication over semirings and offers an interface similar to GraphBLAS. Mapping MAGiQ's operations to CombBLAS operations is straightforward and similar to the mappings to GraphBLAS functions. However, this library also provides an operation that can be used to implement selection operations faster than matrix multiplication: `DimAppy`($\mathbf{M}$, $\mathbf{v}$, f). This operation accepts a sparse matrix $\mathbf{M}$, a vector $\mathbf{v}$, and a binary function f, and produces a sparse matrix. `DimApply` works in two modes 'Column', and 'Row'. When in 'Column' mode, it places the output of $f(\mathbf{M}(i,j), \mathbf{v}(i))$ for all values of i in column j in the output matrix. We use `DimApply` to implement selection operations by passing the RDF matrix $\mathbf{A}$ (or its transpose $\mathbf{A}'$), the resulting vector from `any`($\mathbf{S}$) $* p$, and the operator `iseq`.

6 Post-processing and Result Generation

Once the binding matrices of all query edges are generated, the result generation phase starts (see Algorithm Result-Generation). Binding matrices are processed in LIFO order. Function `Get-IJ`($\mathbf{M}$) (Line 3) retrieves all the non-zero indices from matrix $\mathbf{M}$ in a two-column $\langle$*rowIndex*, *columnIndex*$\rangle$ table. Function `JOIN`(T_1, T_2) in Line 4 corresponds to the usual relational join of tables T_1 and T_2 on the common attribute (i.e., natural join).

In cyclic queries, a binding matrix involving a shadow variable[5] results in a filtering step (Line 5) in addition to `JOIN`. `FILTER` removes tuples if the binding of the original variable and the binding of its shadow variable do not match. The output of Result-Generation is a table with as many columns as the number of variables in the query; each tuple contains valid bindings for each query variable.

Example: Consider the example query graph in Figure 5a. First, we convert it to an acyclic query (Figure 5c). Then, we translate it using Algorithm Query-Translation to the following program, assuming traversal starts from node x:

$$\begin{aligned}\mathbf{M}_{xy} &= \mathbf{I} * b \otimes \mathbf{A}\\ \mathbf{M}_{yz} &= \mathtt{diag}(\mathtt{any}(\mathbf{M}'_{xy})) * b \otimes \mathbf{A}\\ \mathbf{M}_{zw} &= \mathtt{diag}(\mathtt{any}(\mathbf{M}'_{yz})) * a \otimes \mathbf{A}'\\ \mathbf{M}_{yz} &= \mathbf{M}_{yz} \times \mathtt{diag}(\mathtt{any}(\mathbf{M}_{zw}))\\ \mathbf{M}_{xy} &= \mathbf{M}_{xy} \times \mathtt{diag}(\mathtt{any}(\mathbf{M}_{yz}))\end{aligned}$$

After executing this program, the contents of the binding matrices are as follows: $\mathbf{M}_{xy} = \{(A, B)\}$, $\mathbf{M}_{yz} = \{(B, D), (B, E)\}$, and $\mathbf{M}_{zw} = \{(D, E), (E, A)\}$. Algorithm Result-Generation processes the binding matrices in the following order: $\mathbf{M}_{xy}$, $\mathbf{M}_{yz}$, and $\mathbf{M}_{zw}$. First, table R is initialized with bindings of variables x and y from $\mathbf{M}_{xy}$ (Line 1):

$$R = \begin{array}{|c|c|}\hline x & y\\ \hline A & B\\ \hline\end{array}\,.$$

Then, table L is constructed from $\mathbf{M}_{yz}$ (Line 3) and joined with R (Line 4) on the common attribute (i.e., variable) y:

$$R = \mathtt{JOIN}\left(\begin{array}{|c|c|}\hline x & y\\ \hline A & B\\ \hline\end{array}, \begin{array}{|c|c|}\hline y & z\\ \hline B & D\\ \hline B & E\\ \hline\end{array}\right) = \begin{array}{|c|c|c|}\hline x & y & z\\ \hline A & B & D\\ \hline A & B & E\\ \hline\end{array}\,.$$

Finally, `JOIN` and `FILTER` are called when $\mathbf{M}_{zw}$ is encountered because $v_2 = w$ is a shadow variable (Lines 4 and 5):

$$R = \mathtt{JOIN}\left(\begin{array}{|c|c|c|}\hline x & y & z\\ \hline A & B & D\\ \hline A & B & E\\ \hline\end{array}, \begin{array}{|c|c|}\hline z & w\\ \hline D & E\\ \hline E & A\\ \hline\end{array}\right) = \begin{array}{|c|c|c|c|}\hline x & y & z & w\\ \hline A & B & D & E\\ \hline A & B & E & A\\ \hline\end{array}\,,$$

and

$$R = \mathtt{FILTER}\left(\begin{array}{|c|c|c|c|}\hline x & y & z & w\\ \hline A & B & D & E\\ \hline A & B & E & A\\ \hline\end{array}\right) = \begin{array}{|c|c|c|}\hline x & y & z\\ \hline A & B & E\\ \hline\end{array}\,.$$

`FILTER` removed tuples where bindings of x and its shadow variable w do not match, and it removed the w column.

Solution modifiers. SPARQL specifications allow using solution modifiers such as filters based on string matching, regular expressions, or inequalities and aggregations on the bindings of query variables using sum, count, min, or max. MAGiQ can handle filters and aggregation constructs during its post-processing phase. If the filter is over a single variable, then it can be handled during the binding matrices generation phase. Otherwise, the filter is applied in the post-processing phase once all binding matrices are ready. Aggregation functions can be handled similarly.

[5]For simplicity, we assume that a binding matrix of an original variable is encountered before the binding matrix of its shadow variable.

Table 2. Datasets statistics in millions (M). #P is the number of unique predicates in a dataset.

Dataset	#Triples (M)	#Nodes (M)	#P
WatDiv-100M	109.23	10.28	85
YAGO2	284.30	60.70	98
WatDiv-1B	1,092.16	97.39	86
LUBM-1B	1,366.71	336,51	18
Bio2RDF	4,287.59	1,135.93	1,714
LUBM-10B	10,677.83	2,628.99	18
LUBM-512B	512,527.41	126,188.23	18

Table 3. Loading times (minutes); n/a: failed to load within 24 hours or crashed while loading.

Dataset	RDF-3X	gStore	Virtuoso	Wukong[8]	**MAGiQ**
WatDiv-100M	18	40	9	4	**1**
YAGO2	78	63	50	9	**3**
LUBM-1B	447	n/a	191	57	**16**
Bio2RDF	n/a	n/a	331	n/a	**92**

7 Experimental Evaluation

Hardware configuration. Single machine experiments were conducted on a Linux machine with two 14-core Intel Xeon E5-2680 CPUs @ 2.4GHz and 512GB RAM. This machine is equipped with a desktop-grade NVIDIA Quadro P6000 GPU with 24GB GDDR5X GPU memory. Distributed-memory experiments were conducted on a Cray XC40 supercomputer with 6,174 compute nodes each with two 16-core Intel CPUs @ 2.3GHz and 128GB RAM.

Datasets. Table 2 summarizes our datasets. We used the LUBM [5] and WatDiv [11] benchmarks to generate synthetic datasets with up to 512 billion triples (LUBM-512B). These benchmarks contain several queries with different complexities and are commonly used in the literature [13, 34, 48, 55]. We also used the real datasets YAGO2 [12] and Bio2RDF [2] with 284 million and 4.3 billion triples, respectively. Datasets with less than 10 billion triples are used in single machine experiments; the larger ones are used in distributed-memory experiments. The queries we used for each dataset are available online[6].

MAGiQ prototypes. We evaluate four versions of MAGiQ with different back-end libraries: SuiteSparse, Matlab-CPU, Matlab-GPU, and CombBLAS. MAGiQ (SuiteSparse) uses the SuiteSparse:GraphBLAS v1.1.0 [8] implementation of GraphBLAS and runs on a single CPU thread. MAGiQ (Matlab-CPU) and MAGiQ (Matlab-GPU) are built on top of Matlab R2017B. MAGiQ (Matlab-GPU) uses multiple CPU threads while MAGiQ (Matlab-GPU) uses a single GPU. Finally, MAGiQ (CombBLAS) uses CombBLAS v1.6.1 [23], which employs MPI in a distributed-memory environment.

Competitors. We compare against a variety of state-of-the-art and established engines in our experiments, including relational, graph-based, and specialized graph processing hardware. Below is a summary of each engine:

RDF-3X [45]: Relational engine that creates exhaustive indices to accelerate its single-threaded join-based query processor. Even though this is a relatively old engine, it holds the record for some queries compared to state-of-the-art engines.

gStore [56]: Graph-based engine that evaluates SPARQL queries using efficient subgraph matching algorithms. This engine uses a single thread.

UrikaGD [9]: A data analytics appliance by Cray, which consists of a graph-optimized hardware with 2TB of global shared-memory and 64 Threadstorm processors with 128 hardware threads per processor, and provides a SPARQL query engine.

Virtuoso [32]: An enterprise grade solution built on top of a hybrid row/column-oriented DBMS. This engine scales to very large graphs and utilizes muti-core CPUs on a single machine.

Wukong[7] [48]: State-of-the-art engine that runs efficiently on a single machine (using multi-threading) and on RDMA-enabled distributed-memory systems. It employs several query planning and graph exploration techniques to achieve good performance for many queries.

AdPart [35]: State-of-the-art distributed-memory SPARQL query engine. It implements a query optimizer that exploits the query structure and hash-based data locality to produce query execution plans with minimal communication. This engine was shown to outperform several distributed-memory engines in a recent study [13].

RDF-3X and gStore use disk to store indices, so we mounted their indices in memory for fairness. The query times reported for each engine are averaged over 5 runs to account for randomness and noise.

7.1 Single Machine Experiments

7.1.1 Data Loading

Table 3 shows the time needed by each engine to load the input RDF dataset: it includes the time to collect statistics, construct various indices, and perform any required preprocessing before answering queries. All existing engines

[6]https://github.com/fjamour/MAGiQ

[7]The authors of this engine also developed a GPU assisted version: Wukong+G [51]. However, its source code was not available to us at the time of writing this paper, so we were not able to use it in our experiments.

[8]Wukong failed to load YAGO2 dataset initially because this dataset has labels that appear as subjects/objects and predicates, which is not supported in Wukong. We filtered out triples containing such labels from YAGO2 with the help of the authors of Wukong to enable loading.

Table 4. Runtimes for *data-intensive* queries on LUBM-1B (sec).

	L1	L2	L3	L7	GeoMean
RDF-3X	901.4	115.6	898.2	426.2	307.6
Virtuoso	25.0	1,268.2	11.7	308.1	103.3
UrikaGD	5.8	2.4	1.9	7.0	3.7
Wukong	11.1	10.3	10.5	42.0	15.0
MAGiQ (SuiteSparse)	173.2	38.0	107.7	155.0	102.4
MAGiQ (Matlab-CPU)	24.9	14.3	6.1	38.2	17.0
MAGiQ (Matlab-GPU)	**3.2**	**2.1**	**1.5**	**5.4**	**2.7**

require significant loading times. Virtuoso is the only competing engine that was able to load all datasets successfully. MAGiQ is considerably faster. For example, for the LUBM-1B dataset, Virtuoso needed more than 3 hours, while MAGiQ loaded the data in 16 minutes. MAGiQ loads the input datasets faster than all competitors (3x to 28x faster) because it does not build explicit indices; its loading time is dominated by the time to read the graph from disk.

7.1.2 Query Evaluation

LUBM-1B dataset. We used the 7 queries from [18], which are used in most of the RDF literature [13, 34, 35, 48, 55, 56]. LUBM queries can be classified into selective and data-intensive based on size of their intermediate results. L4 and L5 are selective star queries[9] that generate small intermediate and final results, and L6 is a very selective simple query. L2 is a reporting star query that generates large intermediate and final results. L1, L3, and L7 are 6-edge queries with large intermediate results. We refer to computationally light queries (i.e., L4, L5, and L6) as selective queries, and to computationally heavy queries (i.e., L1, L2, L3, and L7) as data-intensive queries.

Table 4 shows the runtimes of data-intensive queries for LUBM-1B dataset. For such queries, existing relational engines such as RDF-3X and Virtuoso need to perform expensive joins (i.e., joining potentially unsorted large intermediate results). More recent engines such as Wukong attempt to mitigate the cost of these expensive joins by using graph exploration coupled with aggressive pruning (i.e., full-history pruning [48]). In the context of MAGiQ, intermediate results are processed using heavily optimized matrix operations. The efficient and highly parallel implementation of these matrix operations available in the underlying back-end matrix algebra libraries enables MAGiQ to provide competitive performance for data-intensive queries compared to existing engines. Particularly, our Matlab-GPU implementation outperforms specialized engines by up to two orders of magnitude (compared to RDF-3X) and is at least 1.4x faster (compared to UrikaGD). Compared to the state-of-the-art engine Wukong, MAGiQ (Matlab-GPU) is 5.6x faster on average, while MAGiQ (Matlab-CPU) is 1.13x slower. Note

[9]Star queries are those with a central node connected with one edge to each of several other nodes.

Table 5. Runtimes for selective queries on LUBM-1B (msec).

	L4	L5	L6	GeoMean
RDF-3X	17	10	181	31
Virtuoso	1,941	32	42	137
UrikaGD	1,442	720	1,588	1,181
Wukong	**1.9**	**0.3**	**1**	**1.3**
MAGiQ (SuiteSparse)	55,167	32,737	68,927	49,931
MAGiQ (Matlab-CPU)	4,892	1,095	2,571	2,397
MAGiQ (Matlab-GPU)	2,337	610	698	998

that our matrix algebra based GPU implementation achieves a speedup up to 7.8x compared to Wukong. This is comparable to the speedups achieved by a specialized state-of-the-art GPU assisted engine, Wukong+G [51], which achieves up to 9x speedup compared to Wukong.

Table 5 shows the runtimes of selective queries for LUBM-1B dataset. Selective queries typically touch a small portion of the data and generate small intermediate results; therefore, the indices of RDF-3X, Virtuoso, and Wukong are beneficial. For such queries, retrieving variable bindings of the query edges is the main bottleneck rather than processing the intermediate results like in the case of data-intensive queries. Wukong is by far the fastest engine for this category of queries with a geometric mean of 1.3ms, compared to 31ms for the second fastest engine RDF-3X. Even though MAGiQ (Matlab-GPU) has a geometric mean of less than a second, MAGiQ is not a good choice for solving selective queries compared to Wukong and RDF-3X. However, the fastest engine Wukong used 312GB of memory to load the LUBM-1B dataset, while MAGiQ used 26GB only.

YAGO2 dataset. Since YAGO2 is a real dataset with no benchmark queries, we used the same set of queries (Y1-Y4) defined in [13, 35]. Y1 and Y2 are selective queries that result in a small number of results, while Y3 and Y4 are data-intensive queries that require non-selective object-object joins. Table 6 shows the runtimes for all compared engines. The conclusions are similar to those of LUBM-1B dataset; MAGiQ provides competitive performance compared to specialized engines for data-intensive queries.

WatDiv-100M dataset. The WatDiv benchmark defines 20 query templates [11] classified into four categories: linear (L), star (S), snowflake (F), and complex queries (C). Similarly to [34, 35], we created 20 queries of each query category. Table 7 shows the runtimes; for each query class, we report the runtime geometric mean for each engine. RDF-3X performs best for star (S) queries, while Wukong achieves the best performance for other query categories. For complex (C) queries, both MAGiQ (Matlab-GPU) and MAGiQ (Matlab-CPU) are significantly faster than all other engines, 3x to 10x faster, except for Wukong. Wukong performs well on WatDiv benchmark because the high selectivity of its queries enables its graph exploration engine to touch only small portions of the data [48].

Table 6. Runtimes for YAGO2 queries (msec).

	Y1	Y2	Y3	Y4	Geo. Mean
RDF-3X	51	234,600	9,800	112	1,904
gStore	274	136	8,473	1,053	758
Virtuoso	537	21	9,136	**16**	202
UrikaGD	1,864	1,649	1,523	1,415	1,604
Wukong[10]	**4**	**5**	172	758	**38**
MAGiQ (SuiteSparse)	26,069	33,139	17,331	21,551	23,834
MAGiQ (Matlab-CPU)	118	122	246	111	141
MAGiQ (Matlab-GPU)	54	66	**105**	40	62

Table 7. Runtimes (GeoMean) for WatDiv-100M queries (msec).

	S1-S7	F1-F5	L1-L5	C1-C3
RDF-3X	**11**	32	11	813
gStore	139	187	230	1,154
Virtuoso	22	30	20	1,213
UrikaGD	1,264	1,330	1,743	2,357
Wukong	16	**2**	**1**	**47**
MAGiQ (SuiteSparse)	1,028	2,168	790	5,393
MAGiQ (Matlab-CPU)	25	44	16	234
MAGiQ (Matlab-GPU)	26	48	16	195

Table 8. Runtimes for Bio2RDF queries (msec).

	B1	B2	B3	B4	B5	Geo. Mean
Virtuoso	588	941	**1,656**	298	198	382
UrikaGD	879	**798**	1,832	1,180	947	1,075
MAGiQ (Matlab-CPU)	658	2,674	7,052	**54**	**8**	351
MAGiQ (Matlab-GPU)	**232**	1,111	2,740	186	10	**265**

Bio2RDF dataset. We used the same Bio2RDF queries (B1-B5), extracted from a real query log, as in [35]. B1 contains two triple patterns that require object-object join. B2 and B3 are star queries with different number of triple patterns. B4 has a 2-hop radius while B5 is a very selective star query with only one triple pattern. We show the runtimes in Table 8. Virtuoso and UrikaGD achieve better performance than MAGiQ for star queries (B2 and B3). However, MAGiQ (Matlab-GPU) and MAGiQ (Matlab-CPU) are faster for the rest of the queries. The intermediate binding matrices of B4 are small, resulting in MAGiQ (Matlab-CPU) being more efficient than MAGiQ (Matlab-GPU). For B5, the cost of copying the predicate matrices to the GPU and fetching the binding matrices back to the CPU is not amortized due to the high selectivity of the query. Therefore, MAGiQ (Matlab-CPU) performs better than MAGiQ (Matlab-GPU). However, the geometric mean of MAGiQ (Matlab-GPU) is lower than all other engines.

7.1.3 Query Workload Evaluation

We show in this experiment a breakdown of MAGiQ's runtime for workloads of queries. Two workloads are used in this experiment: 10,000 LUBM benchmark queries on the LUBM-1B dataset and 20,000 WatDiv benchmark queries on the WatDiv-1B dataset. These workloads were defined by the state-of-the-art distributed-memory workload-aware SPARQL query engine [35]. Runtimes are shown for the following steps: pre-processing (i.e., graph loading), main program execution (i.e., binding matrices computation), copying data to GPU memory (for the GPU implementation), and post-processing (i.e., result generation).

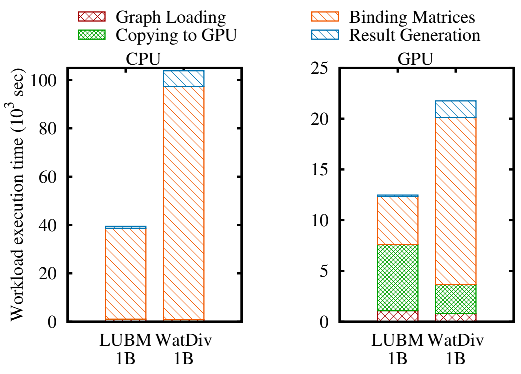
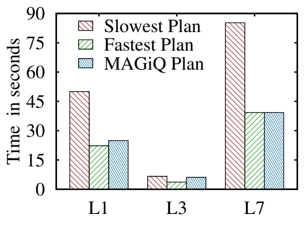


Figure 6. Breakdown of MAGiQ runtime when evaluating 10K and 20K query workloads for LUBM-1B and WatDiv-1B, respectively. Query translation took two seconds. Binding matrices generation is the dominant performance factor.

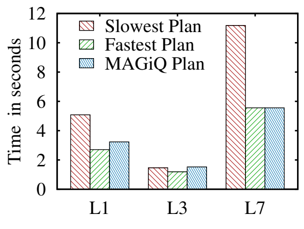
Figure 6 shows the results for the Matlab-CPU and Matlab-GPU implementations. MAGiQ took less than two seconds for translating all the 10K LUBM and the 20K WatDiv queries. For MAGiQ (Matlab-CPU), loading the graph and generating the results incurs insignificant overhead; the dominant factor is the query evaluation program execution. This shows that the performance critical part is the main query program execution, which is affected mainly by the back-end implementation of the matrix algebra operations. Similar observations can be made when running these two workloads using MAGiQ (Matlab-GPU). For the WatDiv-1B workload, the main query program execution dominates the execution time. However, copying LUBM-1B matrices to the GPU took more time compared to WatDiv-1B since LUBM has fewer predicates and more dense predicate matrices.

7.1.4 Effect of Query Planning

In this experiment, we evaluate the efficiency of our query optimizer. We evaluated the LUBM data-intensive queries L1, L3, and L7 using Matlab-CPU and Matlab-GPU; we executed all possible plans for each query. Figure 7 shows the fastest and slowest execution times over all plans. It also compares them with the execution time for the plan selected by MAGiQ. For MAGiQ (Matlab-CPU) (Figure 7a), our optimizer selects a plan that is either optimal in the search space or has performance very close to the fastest execution plan. This is also observed for MAGiQ (Matlab-GPU) (Figure 7b).



(a) MAGiQ (MATLAB-CPU)



(b) MAGiQ (MATLAB-GPU)

Figure 7. Query plan selection for LUBM-1B. MAGiQ's optimizer selects plans close to the optimal.

7.1.5 Data Scalability

In this experiment, we show how the performance of our MATLAB implementations change as we increase the dataset size. We generated 5 datasets ranging from 85 million triples to 1.3 billion triples using the LUBM benchmark. Figure 8a shows the geometric mean of runtimes for queries L1-L7, for each dataset size. As shown, the geometric mean of both CPU and GPU implementations increases slowly with the dataset size.

7.2 Distributed-Memory Experiments

7.2.1 Data Loading and Query Evaluation

We show in this section the performance of MAGiQ (CombBLAS) compared to the performance of the state-of-the-art distributed-memory engine ADPART using the LUBM-10B dataset. Both engines were deployed on our Cray XC40 supercomputer using 1,024 compute nodes.

ADPART took a total of 57.13 minutes to load the graph, while MAGiQ (CombBLAS) took a total of 2.75 minutes (20x faster than ADPART). MAGiQ (CombBLAS) reads the graph once in parallel and distributes it across the available compute nodes. ADPART graph partitioning utility reads the input dataset serially and splits it into one file per compute core, then ADPART query engine loads all the files in parallel. The bottleneck of ADPART is the initial serial graph read, which takes most of the loading time.

Table 9 shows the runtimes for LUBM-10B queries L1-L7. For queries L2-L5, ADPART is faster because it was able to do the evaluation without communication, which was enabled by its data distribution mechanism and locality-aware query planning. For queries L1, L6, and L7, ADPART is slower because it was not able to do communication-free evaluation. MAGiQ (CombBLAS) inherits its efficient communication from CombBLAS, and thus scales to a large number of compute nodes.

7.2.2 Scalability

Figure 8b shows the runtimes of MAGiQ (CombBLAS) for queries L1-L7 on dataset LUBM-10B as we increase the number of compute nodes from 64 to 1,024. CombBLAS has an ideal speedup $\sqrt{p}$ [24], where p is the number of CPU cores.

Table 9. Runtimes for LUBM-10B queries (sec).

	L1	**L2**	**L3**	**L4**	**L5**	**L6**	**L7**
ADPART	5.12	**0.12**	**0.24**	**0.07**	**0.08**	3.51	4.84
MAGiQ (CombBLAS)	**3.08**	0.93	0.67	1.66	0.61	**1.36**	**3.79**

Consequently, the ideal speedup MAGiQ (CombBLAS) is expected to have is $\sqrt{p}$, so we quadruple the number of compute nodes at each step in Figure 8b similarly to [24].

Figure 8c shows the runtimes for queries L1-L7 on 2,048 compute nodes as we increase the dataset size from 64 to 512 billion triples. The increase of runtime is almost linear at such a large scale, which suggests that MAGiQ (CombBLAS) is suitable for querying very large datasets. We used the LUBM benchmark to generate the datasets used in this experiment.

7.3 Discussion and Limitations

While MAGiQ provides competitive performance for data-intensive queries, it is evident in our experimental evaluation that the main limitation is its poor performance for selective queries. Such queries benefit from building exhaustive indices because their evaluation involves selecting very small parts of the input dataset without requiring heavy computations. Consequently, parallelism does not help accelerating such queries. Specialized engines such as Wukong [48] and RDF-3X [45] solve such queries in milliseconds, whereas MAGiQ requires seconds.

In summary, MAGiQ trades off selective query performance for: (*i*) portability over a variety of infrastructures; (*ii*) good performance for data-intensive queries; (*iii*) scalability to very large datasets and computing infrastructures; (*iv*) reduced memory footprint; and (*v*) fast loading time.

8 Related work

Specialized SPARQL engines. Many research efforts focus on building efficient centralized SPARQL query engines [13, 18, 36, 45, 46, 53, 56]. RDF-3X [45] and HexaStore [52] are relational engines that create exhaustive indices to accelerate their join-based query processors. Openlink Virtuoso [32] is a SPARQL engine built on top of a hybrid row/column-oriented DBMS. TripleBit [53] uses compact sorted indices and performs merge-joins for query evaluation. BitMat [18] uses a compressed 3-dimensional bit-matrix for storing RDF graphs and performs joins on the compressed representation for query evaluation without materializing intermediate join results. gStore [56] utilizes the graph storage model to store RDF data and solves SPARQL queries using efficient subgraph matching algorithms. Many distributed engines [13, 14, 34, 35, 42, 47, 48, 51, 55] have also been proposed recently. These engines can be classified into: (*i*) engines built on top of a general purpose data or graph processing system like MapReduce [29], Spark [54], or Pregel [43];

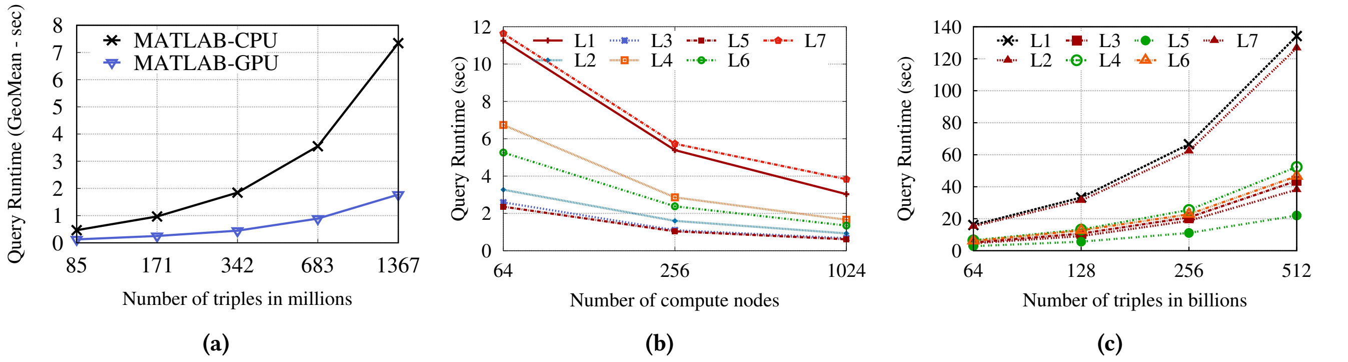


Figure 8. Scalability of MAGiQ: (a) Data scalability on a single machine. (b) Strong scalability on a supercomputer using LUBM-10B. (c) Data scalability on a supercomputer using 2,048 compute nodes.

and (*ii*) engines specifically built for SPARQL query evaluation. Such engines use native RDF indices, efficient communication frameworks, and customized query optimization. All the above mentioned engines are designed with a particular hardware architecture in mind. Therefore, adapting these engines to run effectively on a different architecture, e.g., GPUs, entails almost redesign from scratch. This is evident in Wukong+G [51] which implements several GPU specific optimizations, including GPU-friendly indices, GPU-based query execution, and GPUDirect RDMA. In contrast, MAGiQ is easily portable to a variety of hardware architectures.

Matrix algebra libraries for graphs. GraphBLAS [3] is an emerging standard API of matrix algebra operations for graph processing. The mathematical foundations of GraphBLAS and the motivation for having such a standard interface are described in [39, 40]; the C API of GraphBLAS is described in [25]. SuiteSparse:GraphBLAS provides a fully conformant C implementation of the GraphBLAS C API [8, 27]. Many other graph processing frameworks adopt the philosophy of GraphBLAS by providing a limited set of highly optimized sparse matrix algebra operations. GPI [31] defines an interface very similar to GraphBLAS and provides a distributed implementation built on top of Spark [54]. CombinatorialBLAS [23] provides a scalable distributed C++ implementation of an interface similar to GraphBLAS, optimized for large-scale distributed-memory systems. GraphPad [17] provides another distributed implementation for a subset of GraphBLAS. Some papers propose efficient storage formats for sparse matrices suited for graph processing [20]. The authors in [22, 24] describe doubly compressed sparse column (DCSC) storage format and propose an efficient algorithm for sparse matrix-matrix multiplication using DCSC.

Graph algorithms using matrix algebra. The authors in [49] present an algorithm for computing betweenness centrality [16] using sparse matrix multiplication, and provide some advances in sparse matrix-matrix multiplication. Furthermore, many other graph algorithms have been formulated using matrix algebra, like PageRank, maximum flows, and breadth-first search [23, 41]. The authors of [15] and [50] present implementations of systems that combine the vertex-centric programming paradigm for parallel graph processing [43] with matrix algebra by mapping vertex-centric programs to sparse matrix-vector multiplications. Unlike SPARQL query evaluation, most of these implemented algorithms either have well-defined algebraic formulations or rely on simple graph traversals. To the best of our knowledge, this is the first work to build a graph query engine using matrix algebra. The authors in [44] propose a theoretical framework where RDF graphs are represented as boolean tensors, and SPARQL queries are solved using a mechanism based on Khatri–Rao product, which (quote from the paper) "is relatively unknown and unstudied"; no implementation is provided. Our goal is to build a real framework to be executed in a wide variety of architectures. Since sparse matrix algebra is well studied and efficient libraries are available, we believe it is a good choice for our framework.

9 Conclusions

This paper studies the plausibility of using matrix algebra for building graph query engines by describing MAGiQ and comparing it with specialized graph query engines. MAGiQ leverages the existing rich software infrastructure for processing sparse matrices optimized for many hardware architectures. We presented four prototypes of MAGiQ that run on different hardware: CPUs, GPUs, and distributed-memory systems. MAGiQ combines portability, scalability, and efficiency all together thanks to the sparse matrix algebra design paradigm. It enables almost effortless utilization of different hardware for accelerating query evaluation on RDF data. Our experimental results on various large-scale real and synthetic graphs show that MAGiQ's performance is comparable to or better than state-of-the-art specialized engines for data-intensive queries. Furthermore, we show that MAGiQ scales to very large graphs (4.3 billion edges on a single machine, and 512 billion edges on a distributed-memory machine).

The main limitation of MAGiQ is its poor performance for selective queries compared to index-based approaches. One obvious approach to improve MAGiQ's performance for such queries is to precompute multiplication paths (i.e., common sub-queries). In the context of our matrix algebra approach, this would result in stored binding matrices that can be used while evaluating selective (or even data-intensive) queries that involve precomputed multiplication paths to avoid expensive multiplications. Such approach is employed in some existing RDF query engines [47] and is commonly referred to as precomputed join tables. There are many trade-offs to study if such an approach is implemented, including the time and memory needed to pre-compute and store these binding matrices. In addition to that, the selection of the multiplication paths to compute is non-trivial if general query workloads are to be supported. We plan to explore such ideas in our future work.

Acknowledgments

We thank the anonymous reviewers and our paper's shepherd Jens Teubner for their feedback. We also thank the authors of Wukong [48] for their responsiveness and help with running and understanding their system. For supercomputer time, this research used the resources of the Supercomputing Laboratory at King Abdullah University of Science & Technology (KAUST) in Thuwal, Saudi Arabia.

References

[1] Basic Linear Algebra Subprograms. http://www.netlib.org/blas/.
[2] Bio2RDF Repository. http://bio2rdf.org/.
[3] GraphBLAS Standard. http://graphblas.org.
[4] Intel MKL. https://software.intel.com/mkl.
[5] LUBM Benchmark. http://swat.cse.lehigh.edu/projects/lubm.
[6] NVIDIA cuSPARSE. https://developer.nvidia.com/cusparse.
[7] SPARQL. https://www.w3.org/TR/rdf-sparql-query/.
[8] SuiteSparse. http://faculty.cse.tamu.edu/davis/suitesparse.html.
[9] Urika-GD. http://www.cray.com/sites/default/files/resources/Urika-GD-TechSpecs.pdf.
[10] W3C: RDF. http://www.w3.org/RDF.
[11] WatDiv Benchmark. http://dsg.uwaterloo.ca/watdiv/.
[12] YAGO2 Repository. http://yago-knowledge.org.
[13] Ibrahim Abdelaziz, Razen Harbi, Zuhair Khayyat, and Panos Kalnis. 2017. A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data. *PVLDB* 10, 13 (2017), 2049–2060.
[14] Ibrahim Abdelaziz, Razen Harbi, Semih Salihoglu, and Panos Kalnis. 2017. Combining Vertex-centric Graph Processing with SPARQL for Large-scale RDF Data Analytics. *IEEE TPDS* 28, 12 (2017), 3374–3388.
[15] Yousuf Ahmad, Omar Khattab, Arsal Malik, Ahmad Musleh, Mohammad Hammoud, Mucahid Kutlu, Mostafa Shehata, and Elsayed Tamer. 2018. LA3: A Scalable Link- and Locality-Aware Linear Algebra-Based Graph Analytics System. *PVLDB* 11, 8 (2018), 920–933.
[16] Ziyad AlGhamdi, Fuad Jamour, Spiros Skiadopoulos, and Panos Kalnis. 2017. A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs. In *SSDBM 2017*.
[17] Michael J Anderson, Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Theodore L Willke, and Pradeep Dubey. GraphPad: Optimized Graph Primitives for Parallel and Distributed Platforms. In *IEEE IPDPS 2016*.
[18] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Matrix "Bit" Loaded: a Scalable Lightweight Join Query Processor for RDF Data. In *WWW 2010*.
[19] Scott Beamer, Aydın Buluç, Krste Asanovic, and David Patterson. Distributed Memory Breadth-first Search Revisited: Enabling Bottom-up Search. In *IEEE IPDPSW 2013*.
[20] Maciej Besta, Florian Marending, Edgar Solomonik, and Torsten Hoefler. SlimSell: A Vectorizable Graph Representation for Breadth-First Search. In *IEEE IPDPS 2017*.
[21] Angela Bonifati, Wim Martens, and Thomas Timm. 2017. An Analytical Study of Large SPARQL Query Logs. *PVLDB* 11, 2 (2017), 149–161.
[22] Aydın Buluç and John R Gilbert. On the Representation and Multiplication of Hypersparse Matrices. In *IEEE IPDPS 2008*.
[23] Aydın Buluç and John R Gilbert. 2011. The Combinatorial BLAS: Design, Implementation, and Applications. *IJHPCA* 25, 4 (2011), 496–509.
[24] Aydın Buluç and John R Gilbert. 2012. Parallel Sparse Matrix-matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), 170–191.
[25] Aydın Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *IEEE IPDPSW 2017*.
[26] Chantana Chantrapornchai and Chidchanok Choksuchat. 2018. TripleID-Q: RDF Query Processing Framework using GPU. *IEEE TPDS* 29, 9 (2018), 2121–2135.
[27] Timothy Davis. 2018. Algorithm 9xx: SuiteSparse: GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra. *Submitted to ACM TOMS* (2018).
[28] Timothy Davis. 2018. Graph Algorithms via SuiteSparse: GraphBLAS: Triangle Counting and K-truss. In *IEEE HPEC 2018*.
[29] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *USENIX OSDI 2004*.
[30] Iain S Duff, Roger G Grimes, and John G Lewis. 1989. Sparse Matrix Test Problems. *ACM TOMS* 15, 1 (1989), 1–14.
[31] Kattamuri Ekanadham, William P Horn, Manoj Kumar, Joefon Jann, José Moreira, Pratap Pattnaik, Mauricio Serrano, Gabriel Tanase, and Hao Yu. Graph Programming Interface (GPI): a Linear Algebra Programming Model for Large Scale Graph Computations. In *ACM ICCF 2016*.
[32] Orri Erling. 2012. Virtuoso, a Hybrid RDBMS/Graph Column Store. *IEEE Data Engineering Bulletin* 35, 1 (2012), 3–8.
[33] Mario Arias Gallego, Javier D Fernández, Miguel A Martínez-Prieto, and Pablo de la Fuente. 2011. An Empirical Study of Real-world SPARQL Queries. In *USEWOD workshop 2011*.
[34] Sairam Gurajada, Stephan Seufert, Iris Miliaraki, and Martin Theobald. TriAD: a Distributed Shared-nothing RDF Engine Based on Asynchronous Message Passing. In *ACM SIGMOD 2014*.
[35] Razen Harbi, Ibrahim Abdelaziz, Panos Kalnis, Nikos Mamoulis, Yasser Ebrahim, and Majed Sahli. 2016. Accelerating SPARQL Queries by Exploiting Hash-based Locality and Adaptive Partitioning. *VLDBJ* 25, 3 (2016), 355–380.
[36] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Chen, and Liang Jeff Chen. 2017. Stylus: A Strongly-Typed Store for Serving Massive RDF Data. *PVLDB* 11, 2 (2017).
[37] Fuad Jamour, Ibrahim Abdelaziz, and Panos Kalnis. 2018. A Demonstration of MAGiQ: Matrix Algebra Approach for Solving RDF Graph Queries. *PVLDB* 11, 12 (2018), 1978–1981.
[38] Zoi Kaoudi and Ioana Manolescu. 2015. RDF in the Clouds: A Survey. *VLDBJ* 24, 1 (2015), 67–91.
[39] Jeremy Kepner, Peter Aaltonen, David Bader, Aydın Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical Foundations of the GraphBLAS. In *IEEE HPEC 2016*.
[40] Jeremy Kepner, David Bader, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. 2015. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. *Procedia Computer Science* 51 (2015),

2453–2462.

[41] Jeremy Kepner and John Gilbert. 2011. *Graph Algorithms in the Language of Linear Algebra.* SIAM.

[42] Kisung Lee and Ling Liu. 2013. Scaling Queries over Big RDF Graphs with Semantic Hash Partitioning. *PVLDB* 6, 14 (2013), 1894–1905.

[43] Grzegorz Malewicz, Matthew Austern, Aart Bik, James Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a System for Large-scale Graph Processing. In *ACM SIGMOD 2010*.

[44] Saskia Metzler and Pauli Miettinen. 2015. On Defining SPARQL with Boolean Tensor Algebra. *arXiv preprint arXiv:1503.00301* (2015).

[45] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *VLDBJ* 19, 1 (2010), 91–113.

[46] M Tamer Özsu. 2016. A Survey of RDF Data Management Systems. *Frontiers of Computer Science* 10, 3 (2016), 418–432.

[47] Alexander Schätzle, Martin Przyjaciel-Zablocki, Simon Skilevic, and Georg Lausen. 2016. S2RDF: RDF Querying with SPARQL on Spark. *PVLDB* 9, 10 (2016), 804–815.

[48] Jiaxin Shi, Youyang Yao, Rong Chen, Haibo Chen, and Feifei Li. Fast and Concurrent RDF Queries with RDMA-Based Distributed Graph Exploration. In *USENIX OSDI 2016*.

[49] Edgar Solomonik, Maciej Besta, Flavio Vella, and Torsten Hoefler. Scaling Betweenness Centrality using Communication-Efficient Sparse Matrix Multiplication. In *ACM/IEEE SC 2017*.

[50] Narayanan Sundaram, Nadathur Satish, Md Mostofa Ali Patwary, Subramanya R Dulloor, Michael J Anderson, Satya Gautam Vadlamudi, Dipankar Das, and Pradeep Dubey. 2015. GraphMat: High Performance Graph Analytics Made Productive. *PVLDB* 8, 11 (2015), 1214–1225.

[51] Siyuan Wang, Chang Lou, Rong Chen, and Haibo Chen. Fast and Concurrent RDF Queries using RDMA-assisted GPU Graph Exploration. In *USENIX ATC 2018*.

[52] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB* 1, 1 (2008), 1008–1019.

[53] Pingpeng Yuan, Pu Liu, Buwen Wu, Hai Jin, Wenya Zhang, and Ling Liu. 2013. TripleBit: A Fast and Compact System for Large Scale RDF Data. *PVLDB* 6, 7 (2013), 517–528.

[54] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI 2012*.

[55] Kai Zeng, Jiacheng Yang, Haixun Wang, Bin Shao, and Zhongyuan Wang. 2013. A Distributed Graph Engine for Web Scale RDF Data. *PVLDB* 6, 4 (2013), 265–276.

[56] Lei Zou, Jinghui Mo, Lei Chen, M. Tamer Özsu, and Dongyan Zhao. 2011. gStore: Answering SPARQL Queries via Subgraph Matching. *PVLDB* 4, 8 (2011), 482–493.