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

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

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

Ibrahim Abdelaziz
IBM Research
ibrahim.abdelaziz1@ibm.com

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:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

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, 39, 47].

SPARQL [7] is the de-facto query language and there exist many systems that support SPARQL queries over large RDF datasets [13, 47]. Two design paradigms are dominant: the relational paradigm, which builds exhaustive indices and utilizes relational operators (e.g., joins) to solve SPARQL queries [35, 36, 46]; and the graph-based paradigm, which represents RDF data in its native graph form and uses graph traversals for query evaluation [14, 56, 57]. Regardless of the paradigm, all existing solutions are tightly coupled to a particular hardware architecture, typically CPUs [35, 46, 57]. However, modern computing systems are equipped with diverse hardware configurations; for example, it is common to find GPUs. Unfortunately, few existing SPARQL query engines utilize GPUs [27, 52]. 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.
It turns out there are several research efforts towards this direction [42], 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, 32], betweenness centrality [24, 50], and triangle counting [29]; or are simple graph traversals such as BFS [20]. To the best of our knowledge, there is no implementation of something as complex as a SPARQL query engine.

We propose MAGiQ\textsuperscript{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 operators, 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 [24] 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\textsuperscript{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 [35, 49, 52] 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 $A : \mathbb{Z}^{n \times n}$, where $n$ is the number of nodes in the graph (i.e., the number of unique subjects and objects). A non-zero entry $A(i, j) = p_{ij}$ denotes that subject $i$ is connected to object $j$ with predicate $p_{ij}$; $nnz(A)$ is the number of non-zero entries.

\textsuperscript{1}A demo of MAGiQ was presented in VLDB ’18 [38].

\textsuperscript{2}The trade-off is slower execution for selective queries whose evaluation benefits from indices and generates small intermediate results.
Adding/removing triples to/from the RDF graph corresponds to updating the matrix 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 \((x, y, z, w) \in \{(A, C, D, B)\}\).

Such queries ask for bindings of variables \(v_1 \ldots v_N\) from the input RDF graph; the returned bindings should satisfy the connectivity restrictions defined in the triple patterns \(tp_1 \ldots tp_M\). MAGiQ supports triple patterns with variable/constant subjects and objects, and constant predicates\(^6\). 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 \((?x, b, ?w):\)

\[
\begin{align*}
M_{xy} &= I \ast a \otimes A \\
M_{yz} &= \text{diag}(\text{any}(M_{xy}')) \ast c \otimes A \\
M_{xz} &= M_{xy} \times \text{diag}(\text{any}(M_{yz})) \\
M_{xz} &= \text{diag}(\text{any}(M_{xy})) \ast b \otimes A \\
\end{align*}
\]

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 \(M_{xy}\). The second line uses the bindings of \(y\) and predicate \(c\) to select the bindings of \(z\). The third line updates \(M_{xz}\) to eliminate bindings invalidated by predicate \(c\). Finally, the fourth line uses the bindings of \(x\) in \(M_{xy}\) with predicate \(b\) to select the valid bindings of \(w\).

---

\(\text{SELECT } ?x \ ?y \ ?z \ ?w \text{ WHERE } (\ ?x \langle a \rangle \ ?y . \\
\ ?y \langle c \rangle \ ?z . \\
\ ?x \langle b \rangle \ ?w . \\
)\)

---

\(\text{SELECT } ?v_1 \ldots ?v_N \text{ WHERE } tp_1 \ldots tp_M .\)

\(\text{MAGiQ supports triple patterns with variable/constant subjects and objects, and constant predicates.}\)

\(\text{These queries can be represented as graphs (see Figure 3); we refer to query triple patterns as query edges hereafter.}\)

\(\text{Example: Consider the example SPARQL query in Figure 3. This query is translated into the following matrix algebra program, where query edges are processed in the following order: }\)

\(\text{\(?x, a, ?y\), \(?y, c, ?z\), and \(?x, b, ?w\):}\)

\(\text{\(M_{xy} = I \ast a \otimes A\)}\)

\(\text{\(M_{yz} = \text{diag}(\text{any}(M_{xy}')) \ast c \otimes A\)}\)

\(\text{\(M_{xz} = M_{xy} \times \text{diag}(\text{any}(M_{yz}))\)}\)

\(\text{\(M_{xz} = \text{diag}(\text{any}(M_{xy})) \ast b \otimes A\)}\)

\(\text{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 }\)

\(\text{\(x\) and \(y\) using predicate }\(a\) \text{ from the RDF matrix }\(A\), \text{ and stores }\)

\(\text{the results in matrix }\(M_{xy}\). \text{ The second line uses the bindings of }\)

\(\text{\(y\) and predicate }\(c\) \text{ to select the bindings of }\(z\). \text{ The third line updates }\(M_{xz}\) \text{ to eliminate }\)

\(\text{bindings invalidated by predicate }\(c\). \text{ Finally, the fourth line uses the bindings of }\(x\) \text{ in }\(M_{xy}\) \text{ with }\)

\(\text{predicate }\(b\) \text{ to select the valid bindings of }\(w\).}\)

---

\(\(\text{\(MAGiQ\) allows variable predicates in triple patterns; however, query predicates are typically constants.}\)

\(\text{\(\text{SPARQL specifications also describe other query constructs such as property path expressions and optional statements that we do not cover in this paper.} \text{ We refer interested readers to}\)}\)

\(\text{\(\text{studies of real SPARQL query logs for more information on the significance of the different SPARQL constructs.}\)}\)
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.

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
a & b & c \\
a & b & c \\
a & b & c
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
\]

Semirings. A semiring is a set with two binary operators, “addition” and “multiplication” [42]. 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 1seq as the “multiplication” operator. LOR is logical OR operator and 1seq 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 1seq 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 \(b\).

\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & b
\end{bmatrix}
\otimes
\begin{bmatrix}
a & b & c \\
a & b & c \\
a & b & c
\end{bmatrix}
\begin{bmatrix}
0 & 1 & 0 \\
0 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]

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

Binding matrices. We refer to the matrix that results from a selection operation as binding matrix denoted by \(M_{v_1,v_2} : \mathbb{Z}_2^{n \times n}\). 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 \(M_{v_1,v_2}\) 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 \(\ast\), this operation receives selection matrix \(S : \mathbb{Z}_2^{n \times n}\) and integer \(p\), and produces matrix \(S^p : \mathbb{Z}_2^{n \times n}\) with the 1s of \(S\) multiplied by \(p\).

Matrix multiplication with LOR, 1seq semiring. Denoted by \(\otimes\), this operation receives predicate selection matrix \(S^p : \mathbb{Z}_2^{n \times n}\) and RDF matrix \(A : \mathbb{Z}_2^{n \times n}\), and produces a binding matrix \(M : \mathbb{Z}_2^{n \times n}\).

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

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

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

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

Matrix transposition. Denoted by a single quote (‘), this operation receives matrix \(M : \mathbb{Z}_2^{n \times n}\) and produces the transposed matrix \(M' : \mathbb{Z}_2^{n \times n}\).
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., \( S^a = I \ast a \)).

The variable bindings are stored in binding matrix \( M_{xy} \):

\[
M_{xy} = I \ast a \otimes A
\]

The operation is calculated as follows:

\[
\begin{pmatrix}
A & B & C & D & E \\
A & 1 & 1 & 1 & 1 \\
B & 1 & 1 & 1 & 1 \\
C & 1 & 1 & 1 & 1 \\
D & 1 & 1 & 1 & 1 \\
E & 1 & 1 & 1 & 1
\end{pmatrix}
\]

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 \( M_{v_1v_2} \), the bindings of variable \( v_2 \) can be converted to a selection matrix as follows:

\[
S_{v_2} = \text{diag}(\text{any}(M_{v_1v_2}))
\]

where the any operation returns a vector with all bindings of \( v_2 \) and the \( \text{diag} \) 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 \( p_{v_2v_3} \). The binding matrix \( M_{v_2v_3} \) is computed as follows:

\[
M_{v_2v_3} = S_{v_2} \ast p_{v_2v_3} \otimes A
\]

The bindings in \( M_{v_2v_3} \) 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 \( M_{v_1v_2} \) may have been invalidated by the constraint of edge \((v_2, v_3)\). To accommodate this, \( M_{v_1v_2} \) 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):

\[
M_{v_1v_3} = M_{v_1v_2} \times \text{diag}(\text{any}(M_{v_2v_3}))
\]

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 back-tracking. In DFS-WALK, an edge is labelled forward 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 \( A' \) (Lines 10–12). \( A' \) 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

---

**Table 1. Summary of MAGiQ’s matrix algebra building blocks.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \ast p )</td>
<td>Multiply matrix ( S ) by integer ( p )</td>
</tr>
<tr>
<td>( S \ast p \otimes A )</td>
<td>Select nodes specified in ( S ) with predicate ( p ) from RDF graph ( A )</td>
</tr>
<tr>
<td>( M \times S )</td>
<td>Select nodes specified in ( S ) from binding matrix ( M )</td>
</tr>
<tr>
<td>any(M)</td>
<td>Extract query variable bindings from binding matrix ( M )</td>
</tr>
<tr>
<td>diag(v)</td>
<td>Produce a selection matrix from query variable bindings in ( v )</td>
</tr>
<tr>
<td>one(i)</td>
<td>Produce a binding vector with a single constant ( i )</td>
</tr>
<tr>
<td>( M' )</td>
<td>Transpose matrix ( M )</td>
</tr>
</tbody>
</table>
Algorithm: Query-Translation

Input: Query graph \( q(V_q, E_q) \)
Output: Matrix algebra program: statements \( s_0 \ldots s_k \) each with 4 components \( s_i.M_{inl}, s_i.M_{inr}, s_i.M_{out}, s_i.op \)

\[ \begin{align*}
1 & \text{Let } S = \phi \\
2 & \text{Let } g \text{ be the undirected version of } q \\
3 & \text{qwq} = \text{DFS-walk}(g) \text{ // Forward DFS and backtracking walk} \\
4 & \text{for } e \in \text{qwq} \text{ do} \\
5 & \quad \text{Let } pe \text{ be the edge before } e \text{ in } \text{qwq} \\
6 & \quad \text{Let } p \text{ be the predicate of edge } e \\
7 & \quad \{ x_1, y_2, etype \} = (e, x_1, e, e.type) \\
8 & \quad \{ w_1, w_2 \} = (p.e_{i1}, p.e_{i2}) \\
9 & \quad \text{if } pe.type = \text{‘back’} \text{ then } \{ w_1, w_2 \} = (p.e_{i2}, p.e_{i1}) \\
10 & \quad \text{if } e \text{ is the first edge in } \text{qwq} \text{ then} \\
11 & \quad \quad (M_{inl}, M_{inr}, M_{out}, op) = (M_{inl}, M_{inr}, \emptyset, \emptyset) \\
12 & \quad \quad \text{if } e \notin E_q \text{ then } M_{inl} = A' \\
13 & \quad \quad \text{else if } etype = \text{‘forward’} \text{ then} \\
14 & \quad \quad \quad (M_{inl}, M_{inr}, M_{out}, op) = (M_{inl}, M_{inr}, \emptyset, \emptyset) \\
15 & \quad \quad \quad \text{if } v_1 \neq w_1 \text{ then} \\
16 & \quad \quad \quad \quad (M_{inl} = \text{\text{di}ag}(\text{any}(M_{inl}))) \times p \times A \times \emptyset) \\
17 & \quad \quad \quad \text{if } e \notin E_q \text{ then } M_{inl} = A' \\
18 & \quad \quad \quad \text{else if } etype = \text{‘back’} \text{ then} \\
19 & \quad \quad \quad \quad (M_{inl}, M_{inr}, M_{out}, op) = (M_{inl}, M_{inr}, \emptyset, \emptyset) \\
20 & \quad \quad \quad \quad \text{if } e_{i2} = \text{\text{di}ag}(\text{any}(M_{inl} \in M_{out})), x) \\
21 & \quad \quad \quad s = (M_{inl}, M_{inr}, M_{out}, op) \\
22 & \quad \quad \text{S.append}(s) \\
23 & \text{return } S
\end{align*} \]

resulting binding matrix correspond to nodes with incoming edges with predicate \( p \) in the RDF graph, or rows in \( A' \). Edges in \( \text{qwq} \) with type \text{‘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 \( \text{qwq} \) 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 \( A' \); thus, the row selection is done on \( A' \) (Line 17). Finally, edges with type \text{‘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 \( \text{qwq} \) (as shown in Figure 4): \((x, y, \text{forward}), (y, z, \text{forward}), (y, x, \text{back}), (x, w, \text{forward})\). The first edge translates (Line 11) to a selection of all rows with predicate \( a \): \( M_{xy} = I \times a \otimes A \). After this step, \( M_{xy} \) contains three non-zeros at \((A, C), (A, E), (E, D)\). The second edge is a \text{‘forward’} edge, and the first node in it and \( pe \) are not equal \((v_1 \text{ is } y, \text{ while } w_1 \text{ is } x)\), so the selection matrix \( M_{n_1} \) should have the bindings of \( w_2 = y \), and thus \( M_{xy} \) 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 \text{‘forward’} edges while green arrows are \text{‘back’} edges. \( M_{n_1} = \text{\text{di}ag}(\text{any}(M_{n_1}')) \). The statement for this edge extracts nodes connected to bindings of \( y \) with an outgoing edge from \( y \), or equivalently rows in \( A \) (Lines 14 and 16). The resulting statement is: \( M_{yz} = \text{\text{di}ag}(\text{any}(M_{xy}')) \times c \otimes A \). After this step, \( M_{yz} \) has one non-zero at \((C, D)\). The third edge is a \text{‘back’} edge, which updates the bindings in \( M_{xy} \) to eliminate bindings invalidated by the previous selection (computing \( M_{yz} \)). \( M_{xy} \) is updated by selecting the columns that are bindings of \( y \) in \( M_{yz} \), or \( \text{\text{di}ag}(\text{any}(M_{yz})) \). The resulting statement is (from Line 19): \( M_{xy} = M_{xy} \times \text{\text{di}ag}(\text{any}(M_{yz})) \). After this step, \( M_{xy} \) has one non-zero at \((A, C)\). Finally the last edge is a \text{‘forward’} edge where both \( v_1 \) and \( w_1 \) are equal to \( x \), in which case a row selection is done on \( A \) using bindings of \( x \) in the selection matrix (Line 14), resulting in the following statement: \( M_{wx} = \text{\text{di}ag}(\text{any}(M_{wy})) \times b \otimes A \). After this step, \( M_{wx} \) 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 original 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.
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-zeros 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 $M = S \otimes A$; the transpose $M'$ equals $A' \otimes S'$. Since $S$ is a square diagonal matrix, it is equal to its transpose. Thus, statement $M' = A' \otimes S$ 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 $\text{GrB\_Matrix}$ and $\text{GrB\_Vector}$ data types for storing matrices and vectors. RDF graph construction is done with $\text{GrB\_Matrix\_build}$, which constructs a $\text{GrB\_Matrix}$ from ($\text{rowIndex}$, $\text{columnIndex}$, value) tuples. Since this library supports the GraphBLAS standard, mapping our operations is straightforward. $\times$ and $\otimes$ map to $\text{GrB\_mxm}$ with a $\text{GxB\_PLUS\_TIMES}$ and $\text{GxB\_LOR\_EQ}$ semirings, respectively. Scalar multiplication and $\text{diag}$ map to calls to $\text{GrB\_Matrix\_setElement}$ in $\text{GrB\_NONBLOCKING}$ mode, which avoids reconstructing the matrix upon each call. Any maps to $\text{GrB\_Matrix\_reduce\_Monoid}$ with $\text{GxB\_LOR\_BOOL\_MONOID}$ binary operator. Operation one maps to $\text{GrB\_Vector\_setElement}$. Transposing a matrix maps to $\text{GrB\_transpose}$. 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 $A'$. 

\begin{figure}[t]
\centering
\includegraphics[width=\textwidth]{diagram.png}
\caption{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 $x$.}
\end{figure}
Algorithm: Result-Generation

Input: Binding matrices $M_{v_1}^{e_1}, \ldots, M_{v_2}^{e_2}$
$M_{v_1}^{e_1}$ is the binding matrix for edge $e_1$ in the undirected query graph $g$, which has the bindings of $e_1,v_1$ and $e_2,v_2$
Output: Query results table $R$
1. $R = \text{Get-1J}(M_{v_1}^{e_1})$
2. for $M_{v_1}^{e_1} \in M_{v_1}^{e_1}, \ldots, M_{v_2}^{e_2}$ do
3. \hspace{1em} $L = \text{Get-1J}(M_{v_2}^{e_2})$
4. \hspace{1em} $R = \text{JOIN}(R, L)$
5. \hspace{1em} if $v_2$ is a shadow variable then $R = \text{FILTER}(R)$
6. return $R$

MAGiQ (MATLAB). MATLAB, like most mature matrix algebra packages, does not support matrix multiplication over semirings. However, the LOR.i.seq 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 $A$ with the MATLAB comparison operator $==$, that is $A_p = (A == p)$. Consequently, a LOR.i.seq semiring multiplication $M_{v_1,v_2} = S \times p \circ A$ can be implemented in MATLAB as: $M_{v_1,v_2} = S \times A_p$. 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 $A$ using MATLAB vectorized operations.

MAGiQ (CombBLAS). This implementation uses CombBLAS [24]: 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: $\text{DimApply}(M, v, f)$. This operation accepts a sparse matrix $M$, a vector $v$, and a binary function $f$, and produces a sparse matrix. $\text{DimApply}$ works in two modes: 'Column', and 'Row'. When in 'Column' mode, it places the output of $f(M(i,j),v(i))$ for all values of $i$ in column $j$ in the output matrix. We use $\text{DimApply}$ to implement selection operations by passing the RDF matrix $A$ (or its transpose $A'$), the resulting vector from any($S) * p$, and the operator $i$.seq.

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-1J($M$) (Line 3) retrieves all the non-zero indices from matrix $M$ in a two-column (rowIndex, columnIndex) 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$^3$ 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 $x$:

$M_{xy} = I \ast b \circ A$
$M_{yz} = \text{diag}(\text{any}(M_{xy})) \ast b \circ A$
$M_{zw} = \text{diag}(\text{any}(M_{yz})) \ast a \circ A'$
$M_{bz} = M_{yz} \times \text{diag}(\text{any}(M_{zw}))$
$M_{xy} = M_{xy} \times \text{diag}(\text{any}(M_{yz}))$

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

$$R = \begin{bmatrix} x & y & A & B \end{bmatrix}$$

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

$$R = \text{JOIN} \left( \begin{bmatrix} x & y & z & B & D & E \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & D & E \end{bmatrix}, \begin{bmatrix} y & z \ \ A & B \end{bmatrix} \right) = \begin{bmatrix} x & y & z & B & D & E \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & D & E \end{bmatrix}$$

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

$$R = \text{JOIN} \left( \begin{bmatrix} x & y & z & w \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & E & A \end{bmatrix}, \begin{bmatrix} z & w \ \ A & B & E & A \end{bmatrix} \right)$$

and

$$R = \text{FILTER} \left( \begin{bmatrix} x & y & z & w \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & D & E \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & E & A \end{bmatrix} \right) = \begin{bmatrix} x & y & z & w \ \ \ \ \ \ \ \ \ \ \ \ \ \ \ A & B & E \end{bmatrix}$$

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.

$^3$For simplicity, we assume that the binding matrix of a shadow variable is encountered after the binding matrix of its original variable.
We also used the real datasets YAGO2 [12] and Bio2RDF [2] with different back-end libraries: SuiteSparse, Matlab with less than 10 billion triples are used in single machine with 284 million and 4.3 billion triples, respectively. Datasets 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, 35, 49, 56]. 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.

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-CPU) uses multiple CPU threads while MAGiQ (MATLAB-GPU) uses a single GPU. Finally, MAGiQ (CombBLAS) uses CombBLAS v1.6.1 [24], 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 [46]: 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 [57]: Graph-based engine that evaluates SPARQL queries using efficient subgraph matching algorithms. This engine uses a single thread.

Virtuoso [33]: An enterprise grade solution built on top of a hybrid row/column-oriented DBMS. This engine scales to very large graphs and utilizes multi-core CPUs on a single machine.

Wukong7 [49]: 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 [36]: 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 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, 35, 49, 56]. 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.

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-CPU) uses multiple CPU threads while MAGiQ (MATLAB-GPU) uses a single GPU. Finally, MAGiQ (CombBLAS) uses CombBLAS v1.6.1 [24], 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 [46]: 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 [57]: Graph-based engine that evaluates SPARQL queries using efficient subgraph matching algorithms. This engine uses a single thread.

Virtuoso [33]: An enterprise grade solution built on top of a hybrid row/column-oriented DBMS. This engine scales to very large graphs and utilizes multi-core CPUs on a single machine.

Wukong7 [49]: 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 [36]: 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 pre-processing before answering queries. All existing engines

7The authors of this engine also developed a GPU assisted version: Wukong+G [52]. 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.

8Wukong 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 2. Datasets statistics in millions (M). #P is the number of unique predicates in a dataset.

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Triples (M)</th>
<th>#Nodes (M)</th>
<th>#P</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM-10M</td>
<td>109.23</td>
<td>10.28</td>
<td>85</td>
</tr>
<tr>
<td>YAGO2</td>
<td>284.30</td>
<td>60.70</td>
<td>98</td>
</tr>
<tr>
<td>LUBM-1B</td>
<td>1,092.16</td>
<td>97.39</td>
<td>86</td>
</tr>
<tr>
<td>LUBM-1B</td>
<td>1,366.71</td>
<td>336.51</td>
<td>18</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>4,287.59</td>
<td>1,135.93</td>
<td>1,714</td>
</tr>
<tr>
<td>LUBM-1B</td>
<td>10,677.83</td>
<td>2,628.99</td>
<td>18</td>
</tr>
<tr>
<td>LUBM-512B</td>
<td>512,327.41</td>
<td>126,188.23</td>
<td>18</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>RDF-3X</th>
<th>gStore</th>
<th>Virtuoso</th>
<th>Wukong8</th>
<th>MAGiQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>LUBM-100M</td>
<td>18</td>
<td>40</td>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>YAGO2</td>
<td>78</td>
<td>63</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>LUBM-1B</td>
<td>447</td>
<td>n/a</td>
<td>191</td>
<td>57</td>
<td>16</td>
</tr>
<tr>
<td>Bio2RDF</td>
<td>n/a</td>
<td>n/a</td>
<td>331</td>
<td>n/a</td>
<td>92</td>
</tr>
</tbody>
</table>

8https://github.com/fjamou/MAGiQ
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 (5x 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, 35, 36, 49, 56, 57]. LUBM queries can be classified into selective and data-intensive based on size of their intermediate results. L4 and L5 are selective star queries 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 [49]). In the context of MAGiQ, intermediate results are processed using heavily optimized matrix operations. The efficient and highly parallel implementation of these 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 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 [52], 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, 36]. 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 [35, 36], we created 20 queries of each query category. Table 7 shows the runtimes; for each query class, we report the average 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 [49].
Table 6. Runtimes for YAGO2 queries (msec).

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

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

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

Table 8. Runtimes for Bio2RDF queries (msec).

<table>
<thead>
<tr>
<th></th>
<th>B1</th>
<th>B2</th>
<th>B3</th>
<th>B4</th>
<th>B5</th>
<th>Geo. Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtuoso</td>
<td>588</td>
<td>941</td>
<td>1,656</td>
<td>298</td>
<td>198</td>
<td>382</td>
</tr>
<tr>
<td>UrikaGD</td>
<td>879</td>
<td>798</td>
<td>1,832</td>
<td>1,180</td>
<td>947</td>
<td>1,075</td>
</tr>
<tr>
<td>MAGiQ (MATLAB-CPU)</td>
<td>658</td>
<td>2,674</td>
<td>7,052</td>
<td>54</td>
<td>8</td>
<td>351</td>
</tr>
<tr>
<td>MAGiQ (MATLAB-GPU)</td>
<td>232</td>
<td>1,111</td>
<td>2,740</td>
<td>186</td>
<td>10</td>
<td>265</td>
</tr>
</tbody>
</table>

Bio2RDF dataset. We used the same Bio2RDF queries (B1-B5), extracted from a real query log, as in [36]. 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.

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.

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 [36]. 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 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),

---

Footnote: Reported results are those for optimal query plans manually generated by the authors of the engine. Feeding the original queries directly to Wukong results in significantly slower performance because Wukong does not support automatic query planning for datasets with a large number of unique predicates like YAGO2.
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).

### 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 $\sqrt{p}$ [25], where $p$ is the number of CPU cores. 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 [25].

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 [49] and RDF-3X [46] solve such queries in milliseconds, whereas MAGiQ needs 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 [18, 37, 46, 47, 53, 54, 57]. TripleBit [54] uses compact sorted indices and performs merge-joins for query evaluation. BitMat [18] uses a compressed 3-dimensional bitmatrix for storing RDF graphs and performs joins on the compressed representation for query evaluation without materializing intermediate join results. Many distributed engines [13, 14, 35, 36, 43, 48, 49, 52, 56] 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 [30], Spark [55], or Pregel [44]; and (ii) engines specifically built for SPARQL query evaluation. Such engines use native RDF indices, efficient communication frameworks, and customized query optimization.
the engines mentioned above 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 [52], 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 [40, 41]; the C API of GraphBLAS is described in [26]. SuiteSparse:GraphBLAS provides a fully conformant C implementation of the GraphBLAS C API [8, 28]. Recent work on efficient sparse matrix-vector multiplication [19] is an example on the attention GraphBLAS primitives have been receiving. Many other graph processing frameworks adopt the philosophy of GraphBLAS by providing a limited set of highly optimized sparse matrix algebra operations. GPI [32] defines an interface very similar to GraphBLAS and provides a distributed implementation built on top of Spark [55]. CombinatorialBLAS [24] 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 [21]. The authors in [23, 25] 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 [50] 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 [24, 42]. The authors of [15] and [51] present implementations of systems that combine the vertex-centric programming paradigm for parallel graph processing [44] 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 [45] 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 [48] 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 precompute 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 Wu Kong [49] 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
[10] W3C: RDF. http://www.w3.org/RDF.
[37] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Yang. Design of the GraphBLAS. In IEEE IPDPSW 2017.
[42] Liang He, Bin Shao, Yatao Li, Huanhuan Xia, Yanghua Xiao, Enhong Yang. Design of the GraphBLAS. In IEEE IPDPSW 2017.
Portable, Scalable and Efficient Query Engines for RDF Graphs


