On Defining SPARQL with Boolean Tensor Algebra

Saskia Metzler
Max-Planck-Institut für Informatik
Saarbrücken, Germany
saskia.metzler@mpi-inf.mpg.de

Pauli Miettinen
Max-Planck-Institut für Informatik
Saarbrücken, Germany
pauli.miettinen@mpi-inf.mpg.de

June 19, 2021

Abstract

The Resource Description Framework (RDF) represents information as subject–predicate–object triples. These triples are commonly interpreted as a directed labelled graph. We propose an alternative approach, interpreting the data as a 3-way Boolean tensor. We show how SPARQL queries – the standard queries for RDF – can be expressed as elementary operations in Boolean algebra, giving us a complete re-interpretation of RDF and SPARQL. We show how the Boolean tensor interpretation allows for new optimizations and analyses of the complexity of SPARQL queries. For example, estimating the size of the results for different join queries becomes much simpler.

1 Introduction

The Resource Description Framework (RDF)[1] is a W3C standard for representing information in the web. RDF data consist of subject-predicate-object (s, p, o) triples that are commonly treated as a directed labelled graph. Edges in the RDF graph go from subjects to objects and predicates form the edge labels. The RDF data can be queried with the SPARQL query language[2]. The labelled directed graph interpretation of RDF data allows for defining some SPARQL operations in an intuitive way, but it is not always the most convenient theoretical framework to work with: for example, the (s, p, o) triples treat the predicate in no different way to the subject or object, yet in the graph interpretation, the predicate acts as the edge, while the subject and object are nodes.

In this paper we propose to approach the RDF data as a 3-way Boolean tensor, defining the SPARQL operations using elementary tensor and matrix operations. While seeing RDF data as tensor is nothing new, to the best of our knowledge we present the first comprehensive analysis of SPARQL in terms of Boolean tensor operations (although there are prior work on efficiently implementing specific SPARQL operations using binary tensors, see Section 7).

We want to emphasize that we do not propose that efficient RDF databases should be build on top of the tensor interpretation. We do think, however, that the tensor interpretation makes certain optimization techniques more intuitive than the graph interpretation. This will hopefully yield concrete benefits for the query processing, for example, by providing more efficient cardinality estimators for joins.

After a brief introduction to tensor notation and terminology (Section 2), we will explain the data model and how the SPARQL queries can be evaluated directly using tensor algebra (Sections 3 and 4). For the sake of clarity, we present the correspondence between SPARQL and tensor algebra using examples. We will then start studying the results we can get using our tensor formulation. First we will study the computation of joins and the estimation of their cardinalities (Section 5), before moving to tensor decompositions (Section 6) and their properties. Throughout these two sections, we list a number of propositions. Many of them are either straight forward, or have been proved earlier. Our goal in presenting these results is to show what kind of results our framework facilitates. We will cover some related work, interesting future directions, and conclusions in the last three sections.

[1] http://www.w3.org/TR/rdf-syntax-grammar
[2] http://www.w3.org/TR/rdf-sparql-query

2 Notation

Throughout this paper, we indicate vectors as bold type lower-case letters ($\boldsymbol{v}$), matrices as bold upper-case letters ($\boldsymbol{M}$), and tensors as bold upper-case calligraphic letters ($\boldsymbol{\mathcal{T}}$).

Element (i, j, k) of a 3-way tensor $\boldsymbol{\mathcal{X}}$ is denoted as x_{ijk}. A colon in a subscript denotes taking that mode entirely; for example, $\boldsymbol{X}_{::k}$ is the kth *frontal slice* of $\boldsymbol{\mathcal{X}}$ ($\boldsymbol{X}_k$ in short), and $\boldsymbol{x}_{:jk}$ is a *tube* along the first way of $\boldsymbol{\mathcal{X}}$.

For a 3-way tensor $\boldsymbol{\mathcal{X}}$, $x_{:jk}$ is the *mode-1 (row) fibre*, $x_{i:k}$ is the *mode-2 (column) fibre*, and $x_{ij:}$ is the *mode-3 (tube) fibre*. Furthermore, $\boldsymbol{X}_{::k}$ is the kth frontal slice of $\boldsymbol{\mathcal{X}}$.

A tensor can be *unfolded* into a matrix by arranging its fibres as columns of a matrix. The mode-i *matricization*, of n-by-m-by-l tensor $\boldsymbol{\mathcal{T}}$, denoted $\boldsymbol{T}_{(n)}$, takes the mode-i fibres of $\boldsymbol{\mathcal{T}}$ and arranges them as the columns of matrix $\boldsymbol{T}_{(i)}$. For example, in mode-1 unfolding, the columns of $\boldsymbol{\mathcal{T}}$ constitute the columns of $\boldsymbol{T}_{(1)}$ that has n rows and ml columns.

The outer product of vectors in N modes is denoted by $\boxtimes$. That is, if $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$ are vectors of length m, n, and l, respectively, $\boldsymbol{\mathcal{X}} = \boldsymbol{a} \boxtimes \boldsymbol{b} \boxtimes \boldsymbol{c}$ is an m-by-n-by-l tensor with $x_{ijk} = a_i b_j c_k$.

The *Boolean tensor sum* of binary tensors $\boldsymbol{\mathcal{X}}$ and $\boldsymbol{\mathcal{Y}}$ is defined as $(\boldsymbol{\mathcal{X}} \vee \boldsymbol{\mathcal{Y}})_{ijk} = x_{ijk} \vee y_{ijk}$. For binary matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ where $\boldsymbol{X}$ has r columns and $\boldsymbol{Y}$ has r rows their *Boolean matrix product*, $\boldsymbol{X} \circ \boldsymbol{Y}$, is defined as $(\boldsymbol{X} \circ \boldsymbol{Y})_{ij} = \bigvee_{k=1}^{r} x_{ik} y_{kj}$.

Let $\boldsymbol{X}$ be an n_1-by-m_1 matrix and $\boldsymbol{Y}$ be an n_2-by-m_2 matrix. Their *Kronecker (matrix) product* is the $n_1 n_2$-by-$m_1 m_2$ matrix $\boldsymbol{X} \otimes \boldsymbol{Y}$ defined by

$$\boldsymbol{X} \otimes \boldsymbol{Y} = \begin{pmatrix} x_{11}\boldsymbol{Y} & x_{12}\boldsymbol{Y} & \cdots & x_{1m_1}\boldsymbol{Y} \\ x_{21}\boldsymbol{Y} & x_{22}\boldsymbol{Y} & \cdots & x_{2m_1}\boldsymbol{Y} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n_1 1}\boldsymbol{Y} & x_{n_1 2}\boldsymbol{Y} & \cdots & x_{n_1 m_1}\boldsymbol{Y} \end{pmatrix}. \tag{1}$$

The *Khatri–Rao (matrix) product* of $\boldsymbol{X}$ and $\boldsymbol{Y}$ is defined as "column-wise Kronecker". That is, $\boldsymbol{X}$ and $\boldsymbol{Y}$ must have the same number of columns ($m_1 = m_2 = m$), and their Khatri–Rao product $\boldsymbol{X} \odot \boldsymbol{Y}$ is the $n_1 n_2$-by-m matrix defined as

$$\begin{aligned} \boldsymbol{X} \odot \boldsymbol{Y} &= \begin{pmatrix} \boldsymbol{x}_1 \otimes \boldsymbol{y}_1, \boldsymbol{x}_2 \otimes \boldsymbol{y}_2, \ldots, \boldsymbol{x}_m \otimes \boldsymbol{y}_m \end{pmatrix} \\ &= \begin{pmatrix} x_{11}\boldsymbol{y}_1 & \cdots & x_{1m}\boldsymbol{y}_m \\ x_{21}\boldsymbol{y}_1 & \cdots & x_{2m}\boldsymbol{y}_m \\ \vdots & \ddots & \vdots \\ x_{n_1 1}\boldsymbol{y}_1 & \cdots & x_{n_1 m}\boldsymbol{y}_m \end{pmatrix}. \end{aligned} \tag{2}$$

Notice that if $\boldsymbol{X}$ and $\boldsymbol{Y}$ are binary, so are $\boldsymbol{X} \otimes \boldsymbol{Y}$ and $\boldsymbol{X} \odot \boldsymbol{Y}$.

3 Data Model

Given an RDF graph T, let $S(T)$, $P(T)$, and $O(T)$ denote the sets of distinct subjects, predicates and objects respectively. The number of distinct subjects is denoted by $|S(T)|$, or shorthand $|S|$. Correspondingly $|P(T)| = |P|$ and $|O(T)| = |O|$ denote the number of predicates and objects. The shorthand notation we use in unambiguous cases, for instance a maximal index occurring in a subscript, and if the cardinality is common to all RDF graphs under discussion.

To represent RDF data by a binary tensor, we enumerate all subjects, predicates, and objects to obtain mappings from the items in $S(T)$, $P(T)$, and $O(T)$ to indices. Let s_i denote the ith subject with the corresponding index $i = 1, \ldots, |S|$ and respectively p_j the jth predicate with $j = 1, \ldots, |P|$ and o_k the kth object with $k = 1, \ldots, |O|$.

With this mapping we can represent any RDF graph T as a 3-way binary $|S|$-by-$|P|$-by-$|O|$ tensor $\boldsymbol{\mathcal{T}}$. An element (i, j, k) of $\boldsymbol{\mathcal{T}}$ is 1 if and only if the respective subject-predicate-object triple (s_i, p_j, o_k) is present in the RDF graph T.

4 SPARQL Queries

Simple SPARQL queries consist of two parts, a `SELECT` clause and a `WHERE` clause. The `SELECT` clause identifies the variables to appear in the query result. The `WHERE` clause provides the basic graph pattern to match against the RDF data. A basic graph pattern is a set of triple patterns that matches a subgraph of the RDF data. Triple patterns are subject-predicate-object triples with the option that variables can be placed instead of each specific subject, predicate, or object.

An example for a simple SPARQL query that has a single triple pattern as basic graph pattern is `SELECT * WHERE {?a` T`:`$\mathtt{p}_j$ T`:`$\mathtt{o}_k$`}`. The keyword `SELECT` acts as a projection operator. It identifies the variables to appear in the query result. In this case, we ask for all variables that have been defined. The triple pattern in the `WHERE` part of the query has a variable for the subject, `?a`, indicated by the prepended `?` and a fixed predicate and object from RDF graph T, $\mathtt{p}_j$ and $\mathtt{o}_k$. It matches all RDF triples of T that have predicate $\mathtt{p}_j$ and object $\mathtt{o}_k$.

If the RDF data is represented as a binary 3-way tensor $\boldsymbol{\mathcal{T}}$, the triple pattern selects the fibre $\boldsymbol{t}_{:jk}$. That is a vector of all subjects with predicate j and object k. This vector has a 1 at positions i that correspond to an RDF triple (s_i, p_j, o_k) present in the RDF graph T.

A slice of $\boldsymbol{\mathcal{T}}$ would be selected if only one mode was fixed by the query. A query `SELECT * WHERE` `{?a` T`:`$\mathtt{p}_j$ `?b}` for instance resembles $\boldsymbol{T}_{:j:}$. If we change the projection to `SELECT ?a` we would get the same number of results but only the indices $\mathtt{s}_i$ of the non-zero positions in the slice would appear in the final solution sequence and the indices $\mathtt{o}_k$ remain hidden.

4.1 Basic Graph Patterns – Join

The basic graph pattern where a set of triple patterns must match can be understood as a join operation. Consider for example a basic graph pattern consisting of two triple patterns, `{?a` T`:`$\mathtt{p}_i$ `?b}` `.` `{?c` U`:`$\mathtt{p}_j$ `?b}`. This queries for all RDF triples where the object `?b` is linked to a subject by predicate $\mathtt{p}_i$ of RDF graph T as well as by predicate $\mathtt{p}_j$ of RDF graph U, where $i \in \{1, \ldots, |P(T)|\}$ and $j \in \{1, \ldots, |P(U)|\}$.

Note that the braces in this example are only used to increase readability and could be omitted. For more complex queries however, curly braces define group graph pattern and hence the processing order.

In Boolean tensor algebra, the triple patterns from the example above resemble the slices $\boldsymbol{T}_{:i:}$ and $\boldsymbol{U}_{:j:}$ of RDF tensors $\boldsymbol{\mathcal{T}}$ and $\boldsymbol{\mathcal{U}}$. A join operation on equal objects is equivalent to the Khatri–Rao product of $\boldsymbol{T}_{:i:}$ and $\boldsymbol{U}_{:j:}$. However, in order to compute the Khatri–Rao product, the length of the columns of both slices must match and to obtain a meaningful result, the labels must be in the same order.

Therefore, in case the objects of $\boldsymbol{\mathcal{T}}$ and $\boldsymbol{\mathcal{U}}$ do not map one-to-one, all-zero slices associated with the object labels from $\boldsymbol{\mathcal{T}}$ that have no correspondent object label in $\boldsymbol{\mathcal{U}}$ are appended to $\boldsymbol{\mathcal{U}}$ and vice versa, such that $|O(T)| = |O(U)|$. Furthermore the labels of the objects in $\boldsymbol{\mathcal{T}}$ and $\boldsymbol{\mathcal{U}}$ need to be in a common order.

The result to the basic graph pattern is a matrix of size $|S(T)|\,|S(U)|$-by-$|O|$,

$$\begin{aligned}\boldsymbol{T}_{:i:} \odot \boldsymbol{U}_{:j:} &= \left(\boldsymbol{t}_{:i1} \otimes \boldsymbol{u}_{:j1}, \boldsymbol{t}_{:i2} \otimes \boldsymbol{u}_{:j2}, \ldots, \boldsymbol{t}_{:i|O|} \otimes \boldsymbol{t}_{:u|O|}\right) \\ &= \begin{pmatrix} t_{1i1}\boldsymbol{u}_{:j1} & {}_{1i2}\boldsymbol{u}_{:j2} & \cdots & t_{1i|O|}\boldsymbol{u}_{:j|O|} \\ t_{2i1}\boldsymbol{u}_{:j1} & t_{2i2}\boldsymbol{u}_{:j2} & \cdots & t_{2i|O|}\boldsymbol{u}_{:j|O|} \\ \vdots & \vdots & \ddots & \vdots \\ t_{|S|i1}\boldsymbol{u}_{:j1} & t_{|S|i2}\boldsymbol{u}_{:j2} & \cdots & t_{|S|i|O|}\boldsymbol{u}_{:j|O|} \end{pmatrix} . \end{aligned} \tag{3}$$

This matrix has non-zeroes where objects have corresponding subjects when the predicate is $\mathtt{p}_i$ as well as $\mathtt{p}_j$. It can be regarded as $|S(T)|$ blocks of $|S(U)|$-by-$|O|$ matrices stacked on top of each other. Each block then corresponds to a subject `?a` from T and each row per block corresponds to a subject `?c` from U. This view enables a straight forward conversion from the Khatri–Rao product to the RDF triples: Instead of numbering the row indices from 1 to $|S(T)|\,|S(U)|$, we refer to each row by its block index and the row index within the block. These indices link to both subjects and the corresponding object is encoded by the column index.

There are more options to join triple patterns. Instead of binding the objects, we could bind on the subjects as in `{?a` T`:`$\mathtt{p}_i$ `?b}` `.` `{?a` U`:`$\mathtt{p}_j$ `?c}`, or on both as in `{?a` T`:`$\mathtt{p}_i$ `?b}` `.` `{?a` U`:`$\mathtt{p}_j$ `?b}`, or even on none as in `{?a` T`:`$\mathtt{p}_i$ `?b}` `.` `{?c` U`:`$\mathtt{p}_j$ `?d}`, or the subject of one pattern with the object of the other as in `{?a` T`:`$\mathtt{p}_i$ `?b}` `.` `{?c` U`:`$\mathtt{p}_j$ `?a}`, and so on. Also, it is possible to fix not the predicates but

the subjects or objects, or even both of them or none. In the remainder of this section, we examine these options is detail and see how their Boolean tensor algebra counterparts look like. For this we assume the preprocessing step of matching the labels in the modes to be joined to be already done.

4.1.1 One Each Fixed, One Bound

The first option for joining two triple patterns we examine is where one variable in each pattern is fixed, like the predicate in the example above, and one is bound. The bound variable is indicated by a common name in both triple patterns of the query. To be part of the result RDF triples from both triple patterns must agree on the value of the bound variable. This means, we obtain all triple patterns that have a common label in the dimension of the bound variable.

Suppose we fix the predicates $T{:}\mathtt{p}_i$ and $U{:}\mathtt{p}_j$ in both the triple patterns, where $i \in \{1, \ldots, |P(T)|\}$ and $j \in \{1, \ldots, |P(U)|\}$. This leaves us with four options how to treat the subject and the object:

1. {?a $T{:}\mathtt{p}_i$?b} . {?a $U{:}\mathtt{p}_j$?c}
2. {?a $T{:}\mathtt{p}_i$?b} . {?c $U{:}\mathtt{p}_j$?b}
3. {?a $T{:}\mathtt{p}_i$?b} . {?c $U{:}\mathtt{p}_j$?a}
4. {?a $T{:}\mathtt{p}_i$?b} . {?b $U{:}\mathtt{p}_j$?c}

As well as the predicate, we can fix either the subjects or the objects or a combination of them. This gives us nine more options for each of the above combinations. So, in total there are 36 different ways to perform a join with one variable bound and one in each pattern fixed. When calculating the Khatri–Rao product, the cases we need to distinguish however are less. Fixing a variable corresponds to selecting a slice of the RDF tensor. A slice is a matrix and hence we only care whether the join should be performed on the rows or columns of either matrix. The case where the columns of both matrices are joined, we have already seen in the example above. The case of joining the rows of both matrices can be accomplished by using the transpose of the matrices, and transposing the Khatri–Rao product. Thus, a join on the subjects while the predicates are fixed amounts to

$$\begin{aligned}\left((\boldsymbol{T}_{:i:})^T \odot (\boldsymbol{U}_{:j:})^T\right)^T &= \left(\boldsymbol{t}_{1i:} \otimes \boldsymbol{u}_{1j:}, \boldsymbol{t}_{2i:} \otimes \boldsymbol{u}_{2j:}, \ldots, \boldsymbol{t}_{|O|i:} \otimes \boldsymbol{u}_{|O|j:}\right)^T \\ &= \begin{pmatrix} t_{1i1}\boldsymbol{u}_{1j:} & t_{2i1}\boldsymbol{u}_{2j:} & \cdots & t_{|O|i1}\boldsymbol{u}_{|O|j:} \\ t_{1i2}\boldsymbol{u}_{1j:} & t_{2i2}\boldsymbol{u}_{2j:} & \cdots & t_{|O|i2}\boldsymbol{u}_{|O|j:} \\ \vdots & \vdots & \ddots & \vdots \\ t_{1i|S|}\boldsymbol{u}_{1j:} & t_{2i|S|}\boldsymbol{u}_{2j:} & \cdots & t_{|O|i|S|}\boldsymbol{u}_{|O|j:} \end{pmatrix}^T . \end{aligned} \tag{4}$$

Analogously, to perform a join between the columns of the first matrix and the rows of the second, we compute $(\boldsymbol{T}_{:i:}) \odot (\boldsymbol{U}_{:j:})^T$ and obtain an $|S(T)|\,|O(U)|$-by-$|S(U)|$ matrix where we interpret the rows as $|S(T)|$ blocks of $|O(U)|$ objects. Note that in order compute the Khatri–Rao product, the number of columns (i.e. the dimensions on which we join) must match and hence we require $|S(U)| = |O(T)|$ in this case.

Likewise, to join between the rows of the first and the columns of the second, we need to evaluate $(\boldsymbol{T}_{:i:})^T \odot \boldsymbol{U}_{:j:}$. This yields an $|O(T)|\,|S(U)|$-by-$|S(T)|$ matrix where we interpret the rows as $|O(T)|$ blocks of $|S(U)|$ subjects.

4.1.2 Two Each Fixed, One Bound

As soon as two variables in a triple pattern are fixed, we select a vector from the corresponding RDF tensor $\boldsymbol{\mathcal{T}}$. The triples described by the pattern {$\mathtt{s}_i$ $T{:}\mathtt{p}_j$?a} for example amount to the non-zero entries in $\boldsymbol{t}_{ij:}$. A query like {$T{:}\mathtt{s}_i$ $T{:}\mathtt{p}_j$?a} . {$U{:}\mathtt{s}_k$ $U{:}\mathtt{p}_l$?a}, where $i \in \{1, \ldots, |S(T)|\}$, $k \in \{1, \ldots, |S(U)|\}$, $j \in \{1, \ldots, |O(T)|\}$, and $l \in \{1, \ldots, |O(U)|\}$, returns those triples where there is a 1 in common positions in both vectors, i.e. $\boldsymbol{t}_{ij:} \wedge \boldsymbol{u}_{kl:}$.

If we interpret the vectors $\boldsymbol{t}_{ij:}$ and $\boldsymbol{u}_{kl:}$ as two $|O|$-by-1 matrices, we can use the Khatri–Rao product $\boldsymbol{t}_{ij:} \odot \boldsymbol{u}_{kl:}$ to express the same operation. Analogously, we can fix any two variables from each triple pattern and bind the remaining one. Furthermore, we can easily join also a triple pattern with two fixed variables with a triple pattern with one bound variable using the Khatri–Rao product. The query {$T{:}\mathtt{s}_i$ $T{:}\mathtt{p}_j$?a} . {?b $U{:}\mathtt{p}_l$?a} for instance would yield a $1 \cdot |S(T)|$-by-$|O|$ result.

4.1.3 One Fixed, One Bound

It is also an option to fix only one variable in one of the triple patterns. Now we examine the Boolean tensor algebra version of a query like {?a T:p$_j$?c} . {?d ?b ?c} with $j \in \{1, \ldots, |P(T)|\}$. This query asks for all triples with predicate T:p$_j$ that have an object common to any other triple in the RDF data. Thus, in case of an RDF tensor representation, this amounts to a column-wise matrix–tensor multiplication. This type of multiplication can be achieved by computing the Khatri–Rao product between the matrix $\boldsymbol{T}_{:j:}$ representing the left triple pattern and each slice $\boldsymbol{U}_{:k:}$ of $\boldsymbol{\mathcal{U}}$ along the predicates $k = 1, \ldots, |P(U)|$ together representing the right triple pattern. The outcome then is a 3-way tensor with $|P(U)|$ slices, each of size $|S(T)|\,|S(U)|$-by-$|O|$.

Using the matricization of $\boldsymbol{\mathcal{U}}$ along the mode to be joined, in this case the object and hence the third mode $\boldsymbol{U}_{(3)}$, we can express the calculation described above in a concise form by

$$\begin{aligned} \boldsymbol{T}_{:j:} \odot \boldsymbol{U}_{(3)} &= \left(\boldsymbol{t}_{:j1} \otimes \boldsymbol{u}_{(3):1}, \boldsymbol{t}_{:j2} \otimes \boldsymbol{u}_{(3):2}, \ldots, \boldsymbol{t}_{:j|O|} \otimes \boldsymbol{u}_{(3):|O|}\right) \\ &= \begin{pmatrix} t_{1j1}\boldsymbol{u}_{(3):1} & t_{1j2}\boldsymbol{u}_{(3):2} & \cdots & t_{1j|O|}\boldsymbol{u}_{(3):|O|} \\ t_{2j1}\boldsymbol{u}_{(3):1} & t_{2i2}\boldsymbol{u}_{(3):2} & \cdots & t_{2i|O|}\boldsymbol{u}_{(3):|O|} \\ \vdots & \vdots & \ddots & \vdots \\ t_{|S|j1}\boldsymbol{u}_{(3):1} & t_{|S|j2}\boldsymbol{u}_{(3):2} & \cdots & t_{|S|j|O|}\boldsymbol{u}_{(3):|O|} \end{pmatrix}, \end{aligned} \tag{5}$$

where $\boldsymbol{U}_{(3)}$ is a matrix of size $|P(U)|\,|S(U)|$-by-$|O|$. The result is thus of size $|S(T)|\,|P(U)|\,|S(U)|$-by-$|O|$ and in order to translate back to an RDF graph, the first dimension is regarded as $|S(T)|$ blocks each with $|P(U)|$ blocks of $|S(U)|$ items. This view enables to address every field in the result matrix with a tuple (s$_i$ p$_j$ s$_k$ o$_l$), where $i \in \{1, \ldots, |S(T)|\}$, $j \in \{1, \ldots, |P(U)|\}$, $k \in \{1, \ldots, |S(U)|\}$, and $l \in \{1, \ldots, |O|\}$. The positions of the non-zeroes addressed in this way answer the RDF query.

Of course, if a similar query is posed with the subjects bound instead of the objects, we need to apply the transpose before computing the Khatri–Rao product, and as well transpose the outcome. (This is similar to what has been discussed in case of one variable fixed in each triple.)

4.1.4 Some Fixed, None Bound

We now examine joins between triple patterns where none of the variables is bound. Technically, these are not any more joins. The simplest such case that is worth looking at is where in each of the triple patterns two variables are fixed. There is also the case of all three variables fixed, but this is no more than to compare whether two RDF triples are equal.

A query with two variables fixed in each triple pattern for example is {T:s$_i$ T:p$_j$?a} . {U:s$_k$ U:p$_l$?b}, where $i \in \{1, \ldots, |S(T)|\}$, $j \in \{1, \ldots, |P(T)|\}$. $k \in \{1, \ldots, |S(U)|\}$, and $l \in \{1, \ldots, |P(U)|\}$. The expected result from that query is a list of all combinations of the triples from the left pattern and the triples from the right pattern. As triples in each of the patterns differ only in their objects, this means we are looking for all possible combinations of objects matching the left triple pattern and objects matching the right triple pattern. These we obtain by taking the outer product of $\boldsymbol{t}_{ij:}$ and $\boldsymbol{u}_{kl:}$, that is $\boldsymbol{t}_{ij:}(\boldsymbol{u}_{kl:})^T$.

Similarly, for a join query with one variable fixed in each triple pattern such as {?a T:p$_j$?b} . {?c U:p$_l$?d} the result is all combinations of subject-object pairs from the left triple pattern with subject-object pairs from the right triple pattern. In terms of Boolean algebra such operations can be expressed using the Kronecker product, a generalization of the outer product,

$$\boldsymbol{T}_{:j:} \otimes \boldsymbol{U}_{:l:} = \begin{pmatrix} t_{1j1}\boldsymbol{U}_{:l:} & t_{1j2}\boldsymbol{U}_{:l:} & \cdots & t_{1j|O|}\boldsymbol{U}_{:l:} \\ t_{2j1}\boldsymbol{U}_{:l:} & t_{2j2}\boldsymbol{U}_{:l:} & \cdots & t_{2j|O|}\boldsymbol{U}_{:l:} \\ \vdots & \vdots & \ddots & \vdots \\ t_{|S|j1}\boldsymbol{U}_{:l:} & t_{|S|j2}\boldsymbol{U}_{:l:} & \cdots & t_{|S|j|O|}\boldsymbol{U}_{:l:} \end{pmatrix}. \tag{6}$$

An even more general notion of the outer product is needed to express a join between two triple patterns when no variable is fixed or bound, as in {?a ?b ?c} . {?d ?e ?f}. This can be accomplished multiplying two tensors, $\boldsymbol{\mathcal{T}} \otimes \boldsymbol{\mathcal{U}}$, as discussed in [5]. This operation, although it is possible appears to have little practical relevance.

4.1.5 None Fixed, Some Bound

The other extreme scenario is to have no fixed variables but varying amounts of bound variables in the triple patterns to join. The last case discussed in the previous section already is an example of such a join. The other end however is much simpler to start with: Consider a join query where all variables are bound. That would be {?a ?b ?c} . {?a ?b ?c}, which is asking for all RDF triples that occur in the RDF graph T and that match all RDF triples that occur in the RDF graph U. So, this query just yields all triples that occur in both T and U. In Boolean tensor algebra, this amounts to the position-wise OR between the corresponding tensors $\boldsymbol{\mathcal{T}}$ and $\boldsymbol{\mathcal{U}}$ of common dimension $|S|$-by-$|P|$-by-$|O|$ (possibly achieved by a preprocessing step),

$$\bigvee_{i=1}^{|S|}\bigvee_{j=1}^{|P|}\bigvee_{k=1}^{|O|} \boldsymbol{t}_{ijk} \otimes \boldsymbol{u}_{ijk}\,, \tag{7}$$

where $i = 1, \ldots, |S|$, $j = 1, \ldots, |P|$, and $k = 1, \ldots, |O|$.

Similarly, if two variables are bound as in {?a ?b ?c} . {?a ?e ?c}), we receive only those triples where the subjects and objects match each other. This means, we take the outer product of each tube $\boldsymbol{t}_{i:k}$ from $\boldsymbol{\mathcal{T}}$ with the corresponding tube $\boldsymbol{u}_{i:k}$ from $\boldsymbol{\mathcal{U}}$,

$$\bigvee_{i=1}^{|S|}\bigvee_{k=1}^{|O|} \boldsymbol{t}_{i:k} \otimes \boldsymbol{u}_{i:k}\,, \tag{8}$$

where $i = 1, \ldots, |S|$, $k = 1, \ldots, |O|$.

In case one variable bound as in {?a ?b ?c} . {?a ?e ?f} we receive only those triples where the subjects match each other. This means that, we take the Kronecker product of each slice $\boldsymbol{T}_{i::}$ from $\boldsymbol{\mathcal{T}}$ with the corresponding slice $\boldsymbol{U}_{i::}$ from $\boldsymbol{\mathcal{U}}$. For $i = 1, \ldots, |S|$, we get

$$\bigvee_{i=1}^{|S|} \boldsymbol{T}_{i::} \otimes \boldsymbol{U}_{i::}\,, \tag{9}$$

a matrix is of size $|S(T)|\,|S(U)|$-by-$|O(T)|\,|O(U)|$.

4.2 Optional Graph Patterns – Left Outer Join

Apart from the basic graph patterns, there are various other ways to combine triple patterns: group graph patterns, optional graph patterns, alternative graph patterns, and patterns on named graphs. Group graph patterns define the evaluation hierarchy which directly translates to associativity of Boolean tensor operations. Alternative graph patterns refer to the union of triple patterns which easily translates to ORing the respective tensor slices. Patterns on named graphs refer to the possibility to join triple patterns from different RDF graphs.

The combination of triple patterns discussed now are the optional graph patterns. These patterns resemble left outer joins. This means, all triples from the left triple pattern have to appear in the final result. If they do not match any triple from the right pattern, they are listed paired with a blank item. Hence, we use all the triples that match, like in the normal join, and some more. To accomplish a left outer join operation using Boolean operations, we can thus use the join operation discussed above, but additionally we need to handle blank items.

Consider the case where we join two triple patterns with fixed predicates and bound subjects (Equation 4). In the Khatri–Rao product, we get a 1 at positions where the left and right triples match with their subjects. Additionally now we need to cover triples with subjects that occur in the left triple pattern but not in the right. To do that we append a column to $\boldsymbol{U}_{:j:}$, the slice corresponding to the right triple pattern. That column has a 1 in rows where $\boldsymbol{U}_{:j:}$ is all-zero and $\boldsymbol{T}_{:i:}$ (the slice corresponding to the left triple pattern) has at least one non-zero. To evaluate {?a T:p$_i$?b} OPTIONAL {?a U:p$_j$?c}, we first append the extra column to $\boldsymbol{U}_{:j:}$ and obtain

$$\boldsymbol{U'}_{:j:} = \left[\boldsymbol{U}_{:j:}, \boldsymbol{k}\right] = \left[\boldsymbol{U}_{:j:}, \bigvee_{r=1}^{|O(U)|} \boldsymbol{u}_{:ir} \wedge \neg \bigvee_{s=1}^{|O(U)|} \boldsymbol{u}_{:js}\right]. \tag{10}$$

Then, like for the join on subjects, we compute the transposed Khatri–Rao product of the transposes of $\boldsymbol{T}_{:i:}$ and $\boldsymbol{U'}_{:j:}$, and receive an $|S|$-by-$|O(T)|\,(|O(U)|+1)$ matrix as the result,

$$\left((\boldsymbol{T}_{:i:})^T \odot (\boldsymbol{U'}_{:j:})^T\right)^T = \left(\boldsymbol{t}_{1i:} \otimes \boldsymbol{u'}_{1j:}, \boldsymbol{t}_{2i:} \otimes \boldsymbol{u'}_{2j:}, \ldots, \boldsymbol{t}_{|O|i:} \otimes \boldsymbol{u'}_{(|O|+1)j:}\right)^T . \tag{11}$$

The additional column we introduce actually stands for "no value" and should be treated accordingly in any follow up calculation on the obtained result.

4.3 Unique Results – Select Distinct

Until now, we discussed different `WHERE` clauses. This section focuses on the keyword `DISTINCT`, a modifier of the `SELECT` clause. This keyword states that the solutions in the result sequence of the query must be unique, hence no duplicate solutions can occur. The naïve way to approach this type of query is to first compute the solution from the `WHERE` clause and then purge the duplicates. Of course, this can also be accomplished similarly treating the RDF data as a binary tensor: Compute the result for the `WHERE` clause using the operations stated above, then OR along the dimensions not asked for in the `SELECT` clause. But the binary tensor representation offers also a more straight forward approach for such queries: For a join query accompanied by `DISTINCT`, instead of computing the Khatri–Rao product and then OR, the result can be obtained immediately.

The first case we examine is a join query where `SELECT DISTINCT` chooses the bound variable, such as `SELECT DISTINCT ?b WHERE {?a` T`:p`$_i$ `?b} . {?c` U`:p`$_j$ `?b}`. Suppose we use the Khatri–Rao product to calculate the result of the `WHERE` clause. Then, the distinct objects `?b` are those corresponding to the non-zero columns,

$$\bigvee_{k=1}^{|S(T)|} (\boldsymbol{T}_{:i:} \odot \boldsymbol{U}_{:j:})_{k:} . \tag{12}$$

In fact, the objects that constitute the result are those which occur at least once in both $\boldsymbol{T}_{:i:}$ and $\boldsymbol{U}_{:j:}$. Hence, this calculation simplifies to evaluating whether both the columns of $\boldsymbol{T}_{:i:}$ and $\boldsymbol{U}_{:j:}$ are non-zero,

$$\left(\bigvee_{k=1}^{|S(T)|} \boldsymbol{t}_{ki:}\right) \wedge \left(\bigvee_{k=1}^{|S(U)|} \boldsymbol{u}_{kj:}\right) . \tag{13}$$

The resulting vector of length $|O|$ has a 1 at positions that refer to objects which are part of the result and zeroes elsewhere.

The next case is to uniquely select pairs of one bound and one free variable, as for instance in `SELECT DISTINCT ?a ?b WHERE {?a` T`:p`$_i$ `?b} . {?c` U`:p`$_j$ `?b}`. To evaluate this query, we first calculate the $|S(T)|\,|S(U)|$-by-$|O|$ Khatri–Rao product as discussed in Section 4.1. Note that we treat the rows of the resulting matrix as $|S(T)|$ blocks, each of $|S(U)|$ columns. The next step then is to get a matrix of size $|S(T)|$-by-$|O|$ that has a 1 at position (m, n) if in the m-th block of the Khatri–Rao product the n-th column has at least one non-zero (with $m \in \{1, \ldots, |S(T)|\}$ and $n \in \{1, \ldots, |O|\}$). More straightforward, the same answer is obtained by taking those columns of $\boldsymbol{T}_{:i:}$ where the corresponding columns of $\boldsymbol{U}_{:j:}$ have at least one non-zero,

$$\boldsymbol{T}_{:i:} \circ \left(\bigvee_{k=1}^{|S(U)|} \boldsymbol{u}_{kj:}\right) . \tag{14}$$

The final case we examine is to uniquely select pairs of the unbound variables, as in the query `SELECT DISTINCT ?a ?c WHERE {?a` T`:p`$_i$ `?b} . {?c` U`:p`$_j$ `?b}`, where $i \in \{1, \ldots, |P(T)|\}$ and $j \in \{1, \ldots, |P(U)|\}$. We show how the naïve way to first compute the Khatri–Rao product followed by OR-ing along the columns corresponds to the Boolean matrix product,

$$\boldsymbol{T}_{:i:} \circ \boldsymbol{U}_{:j:} . \tag{15}$$

The query asks for all unique `?a`–`?c` pairs that match on `?b`. As described in Section 4.1, to get all `?a`–`?c` pairs that match on `?b` we compute the Khatri–Rao product of the respective slices of $\boldsymbol{\mathcal{T}}$ and $\boldsymbol{\mathcal{U}}$. To

retrieve only unique results in terms of subject–subject combinations, we compute OR along the columns of the Khatri–Rao product (corresponding to the objects). This is

$$\bigvee_{k=1}^{|O|} \left(\boldsymbol{T}_{:i:} \odot \boldsymbol{U}_{:j:}\right)_{:k} = \bigvee_{k=1}^{|O|} \left(\boldsymbol{t}_{:i1} \otimes \boldsymbol{u}_{:j1}, \boldsymbol{t}_{:i2} \otimes \boldsymbol{u}_{:j2}, \ldots, \boldsymbol{t}_{:i|O|} \otimes \boldsymbol{u}_{:j|O|}\right)_{:k} \tag{16}$$

$$= \begin{pmatrix} \bigvee_{k=1}^{|O|} & t_{1ik}u_{1jk} \\ \bigvee_{k=1}^{|O|} & t_{1ik}u_{2jk} \\ & \vdots \\ \bigvee_{k=1}^{|O|} & t_{1ik}u_{|S|jk} \\ \bigvee_{k=1}^{|O|} & t_{2ik}u_{1jk} \\ \bigvee_{k=1}^{|O|} & t_{2ik}u_{2jk} \\ & \vdots \\ \bigvee_{k=1}^{|O|} & t_{|S|ik}u_{|S|jk} \end{pmatrix} = \left(\boldsymbol{T}_{:i:} \circ \boldsymbol{U}_{:j:}\right)_{(1)} , \tag{17}$$

the vectorization of the Boolean matrix product $\boldsymbol{T}_{:i:} \circ \boldsymbol{U}_{:j:}$ along the columns. This matches the well-known fact that join-distinct in standard relational databases can be computed using the Boolean matrix product. In particular, we can as well apply the techniques for fast computation of these joins to this case (cf. Section 5.2).

One conceptual difference between first computing the initial solution sequence from the `WHERE` clause and then applying `DISTINCT` over computing the result in one step is that the specified execution order cannot be obeyed. SPARQL defines that `ORDER BY` statements must be applied on the initial solution sequence before the projection. This means, after processing everything from the `WHERE` clause but before anything from the `SELECT` statement. However, by using binary tensors instead of graphs to represent the RDF data, and in particular by using lists of labels instead of sets, we already introduce an ordering even before processing the `WHERE` clause. If this ordering obeys the `ORDER BY` statement, we can also represent the output in the appropriate order without the intermediate step.

Note that SPARQL also defines the keyword `REDUCED` that modifies the `SELECT` clause such that it outputs at most all results that `SELECT` without modifiers would yield and at least those results that `SELECT DISTINCT` yields. Hence, there is no new operation to define for that modifier, as we already discussed two options to provide a valid answer to a `SELECT REDUCED` query.

4.4 Matching Alternatives – Union

The keyword `UNION` may be placed between two graph patterns in a `WHERE` clause. The result of a union query is a concatenation of the solution sequences of the graph patterns. We can interpret this as an OR between the results of both graph patterns. Note however that no matching takes place in this type of query. The initial solution sequence comprises all results from both graph patterns, Each result is equipped with all the variables defined that occur in either of the graph patterns, possibly left blank.

Consider `SELECT ?b WHERE {?a` T`:`$\mathtt{p}_i$ `?b} UNION {?c` U`:`$\mathtt{p}_j$ `?b}`. The initial solution sequence comprises all `?a`–`?b` pairs with an additional blank `?c`, as well as all `?c`–`?b` pairs with an additional blank `?a`. This is the same as concatenating $\boldsymbol{U}_{:i:}$ and $\boldsymbol{T}_{:j:}$ along the columns. The final result would be all objects from T with predicate $\mathtt{p}_i$ and all objects from U with predicate $\mathtt{p}_j$. In our terminology, it would be a vector of length $|O(T)| + |O(U)|$ which has a 1 at positions that either refer to an object of T or U that occurs in the result. Similarly, in a query like `SELECT ?a ?c WHERE {?a` T`:`$\mathtt{p}_i$ `?b} UNION {?c` U`:`$\mathtt{p}_j$ `?d}` $\boldsymbol{U}_{:i:}$ and $\boldsymbol{T}_{:j:}$ would be concatenated along the diagonal, as no variable is bound. The final solution sequence would comprise all subjects from T and all subjects from U, bound to separate variable names, each paired with a blank.

4.5 Further Operations

While join and union operations directly act on the data structure, SPARQL also defines other types of operations that act on the labels of the data. For instance, it is possible to order the solution sequence using a comparator on the labels of one mode. Also, one can use filters to restrict the solutions to those

for which the filter expression yields TRUE. Filters can for example be comparators to restrict numeric values, or regular expressions to restrict the values of strings.

The `ASK` statement can be used in place of `SELECT`. The result of such a query is TRUE if the solution sequence obtained from the `WHERE` part is non-empty. Hence, if we calculate the result on Boolean tensors, we emit TRUE as soon as the first non-zero appears in the result.

SPARQL provides the keyword `CONSTRUCT` that allows generate a new RDF graph from the result of a query. As any RDF graph can be expressed as a Boolean tensor, an RDF tensor can as well be constructed by such a query.

5 Cardinality and Computation of Joins

An important problem in query processing is to estimate the cardinality of join operations. This problem has attracted a significant amount of research in traditional relational databases. For SPARQL joins, Neumann and Moerkotte [21] proposed so called *characteristic sets* to estimate the cardinalities of SPARQL joins (specifically, the type of joins they called *star joins*). In this section we study how the size of join operations can be computed (or estimated) given our framework.

5.1 Khatri–Rao Products

Let us first assume we have stored the marginal sums along each mode of the $|S|$-by-$|P|$-by-$|O|$ data tensor $\boldsymbol{\mathcal{T}}$. That is, we have three matrices, $\boldsymbol{P}$ ($|P|$-by-$|O|$), $\boldsymbol{Q}$ ($|S|$-by-$|O|$), and $\boldsymbol{R}$ ($|S|$-by-$|P|$), for the column, row, and tube marginal sums, respectively. The element (i, j) of $\boldsymbol{Q}$, for example, would be computed as $r_{ij} = \sum_{k=1}^{|P|} t_{ijk}$.

The number of triples returned by a join, with no projection or `DISTINCT` keyword, is the number of non-zeroes in the result. When a join can be expressed as a Khatri–Rao product between two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ (as is the case with most joins considered in Sections 4.1 and 4.2), the number of non-zeroes in $\boldsymbol{A} \odot \boldsymbol{B}$ can be determined exactly using the column marginal sums of $\boldsymbol{A}$ and $\boldsymbol{B}$. Specifically, if $\boldsymbol{A}$ and $\boldsymbol{B}$ both have n columns, let $\boldsymbol{\sigma^A} = (\sigma_i^{\boldsymbol{A}})_{i=1}^n$ and $\boldsymbol{\sigma^A} = (\sigma_i^{\boldsymbol{B}})_{i=1}^n$ be row vectors that contain the column marginals of $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively (e.g. $\sigma_i^{\boldsymbol{A}} = \sum_j a_{ji}$).

Proposition 5.1. Let $\boldsymbol{\sigma^A}$ and $\boldsymbol{\sigma^B}$ be as above. The number of non-zeroes in $\boldsymbol{A} \odot \boldsymbol{B}$ is

$$|\boldsymbol{A} \odot \boldsymbol{B}| = \sum_{i=1}^{n} \sigma_i^{\boldsymbol{A}} \sigma_i^{\boldsymbol{B}} = \boldsymbol{\sigma^A} (\boldsymbol{\sigma^B})^T \,. \tag{18}$$

The column marginal vectors $\boldsymbol{\sigma}$ are naturally just appropriate rows or columns of the tensor marginal sum matrices $\boldsymbol{P}$, $\boldsymbol{Q}$, or $\boldsymbol{R}$. If they are stored in a sparse format, the size of the join can be computed exactly in time $\Theta(\alpha + \beta)$, where α and β are the number of non-empty columns of $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively.

We can also obtain an upper bound for the size of the join in constant time if in addition we store the l_2-norms for each row and column of the marginal sum matrices (that is, $\|\boldsymbol{\sigma}\|$ for every possible $\boldsymbol{\sigma}$):

Proposition 5.2. Let $\boldsymbol{\sigma^A}$ and $\boldsymbol{\sigma^B}$ be as above. Then

$$|\boldsymbol{A} \odot \boldsymbol{B}| \leq \left\|\boldsymbol{\sigma^A}\right\| \left\|\boldsymbol{\sigma^B}\right\| \,. \tag{19}$$

Proof. Noticing that $\boldsymbol{\sigma^A}(\boldsymbol{\sigma^B})^T = \left\|\boldsymbol{\sigma^A}\right\| \left\|\boldsymbol{\sigma^B}\right\| \cos\theta$ together with (18) gives the result as $\cos\theta \leq 1$. □

As of now, writing the join as a Khatri–Rao product does not seem to bring significant benefits on computing the full join, though. The best worst-case bound is $O(|\boldsymbol{A}|\,|\boldsymbol{B}|)$ from the straight-forward evaluation of the algorithm.

As mentioned in Section 4.3, those `SELECT DISTINCT` queries that choose the bound variable can be computed by selecting the non-zero columns of the Khatri–Rao product. Consequently, the cardinality of the result is $\boldsymbol{\sigma^A}(\boldsymbol{\sigma^B})^T$ (taking vectors $\boldsymbol{\sigma}$ as row vectors) and the whole query (not just the cardinality) can be computed in time $\Theta(\alpha + \beta)$.

All of the above computations require an access to the marginal sums. We argue that storing (and updating) this information is feasible for the data base system. In principle, the matrices can be very large, but notice that every (s, p, o) triple of the data has effect to exactly one element in each of the three matrices. Hence, the total number of non-zeroes in the marginal matrices cannot exceed $3\,|\boldsymbol{\mathcal{T}}|$, and in practice it would be expected to stay much smaller.

5.2 Boolean Products

When the `SELECT DISTINCT` query asks for a pair of variables, the evaluation does not (have to) involve the Khatri–Rao product, but rather the Boolean matrix product (Section 4.3). Here, estimating the size of the result becomes harder. Take, for example (16): computing the result involves taking an OR over the columns of a Khatri–Rao product, and hence, even if we know the cardinalities of each column, we can only give very coarse estimations on the final cardinality:

Proposition 5.3. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be two binary matrices with n columns and let $\boldsymbol{\sigma}^{\boldsymbol{A}}$ and $\boldsymbol{\sigma}^{\boldsymbol{B}}$ be their column marginal sum vectors. Then,

$$\max_{i=1}^{n}\{\sigma_i^{\boldsymbol{A}}\sigma_i^{\boldsymbol{B}}\} \leq \left|\bigvee_{i=1}^{n}(\boldsymbol{A}\odot\boldsymbol{B})_{:i}\right| \leq \sum_{i=1}^{n}\sigma_i^{\boldsymbol{A}}\sigma_i^{\boldsymbol{B}} \ . \tag{20}$$

As a Khatri–Rao product, $\boldsymbol{A}\odot\boldsymbol{B}$, contains a specific structure that could help estimating the cardinality better. Another approach is to estimate the cardinality from the Boolean product formulation (17). We will first sketch some estimates on the expectation that are simple and fast to compute, and could therefore be of interest to practitioners.

Proposition 5.4. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be m-by-k and k-by-n binary matrices whose non-zeroes are uniformly distributed, and let $p^{\boldsymbol{A}}$ and $p^{\boldsymbol{B}}$ be the densities of $\boldsymbol{A}$ and $\boldsymbol{B}$, respectively. Then

$$\mathbf{E}[|\boldsymbol{A}\circ\boldsymbol{B}|] = 1-\left(1-p^{\boldsymbol{A}}p^{\boldsymbol{B}}\right)^k \ . \tag{21}$$

We can improve our estimation of the expectation if we notice that the Boolean product is equivalent to element-wise ORs of k rank-1 binary matrices.

Proposition 5.5. Let $\boldsymbol{A}$ and $\boldsymbol{B}$ be m-by-k and k-by-n binary matrices and let $\boldsymbol{p}^{\boldsymbol{A}} = (p_i^{\boldsymbol{A}})_{i=1}^k$ and $\boldsymbol{p}^{\boldsymbol{B}} = (p_i^{\boldsymbol{B}})_{i=1}^k$ be the column densities of $\boldsymbol{A}$ and row densities of $\boldsymbol{B}$, respectively. Assuming the non-zeroes in the rank-1 matrices $\boldsymbol{a}_{:i}\boldsymbol{b}_{i:}$ are distributed uniformly at random for all $i = 1,\dots,k$, we have that

$$\mathbf{E}[|\boldsymbol{A}\circ\boldsymbol{B}|] = 1-\prod_{i=1}^{k}p_i^{\boldsymbol{A}}p_i^{\boldsymbol{B}} \ . \tag{22}$$

Notice that if we take the union bound of the densities of the rank-1 matrices to obtain the upper bound for the number of non-zeroes in $\boldsymbol{A}\circ\boldsymbol{B}$, we obtain the same result as in Proposition 5.3.

The above methods, while being straight forward, require only the marginal sums, which makes them relatively fast to compute. If we can access the whole matrices, we can do much better estimations. In particular, we can use the result of Amossen et al. [1]:

Proposition 5.6 ([1]). There exists an algorithm that obtains an $1+\varepsilon$ approximation of $|\boldsymbol{A}\circ\boldsymbol{B}|$ in expected time $O(|\boldsymbol{A}|+|\boldsymbol{B}|)$ for any $\varepsilon > 4/\sqrt[4]{|\boldsymbol{A}|+|\boldsymbol{B}|}$.

Computing the Boolean product can also be done faster than the standard matrix product: for example, Amossen and Pagh [2] proposed an algorithm that runs in time $\tilde{O}(s^{2/3}z^{2/3}+s^{0.862}z^{0.408})$, where $s = |\boldsymbol{A}|+|\boldsymbol{B}|$ is the number of non-zeroes in the input, $z = |\boldsymbol{A}\circ\boldsymbol{B}|$ is the number of non-zeroes in the output, and $\tilde{O}(\cdot)$ hides the polylogarithmic factors. For sparse input matrices, the bound is generally very good, but if the input matrices are quite dense (and the output moderately so), the overall time complexity exceeds that of the standard matrix multiplication, i.e. $\tilde{O}(m^\omega)$ for m-by-m matrices. To address this, Lingas [18] proposed a randomized algorithm that runs in time $\tilde{O}(m^2 s^{\omega/2-1})$, where the factor matrices are m-by-m and s is as above. (See [18] for an extension to rectangular matrices.)

6 Tensor Decompositions

Similarly to matrices, decomposition is a natural operation to tensors. A tensor decomposition reveals regularities of the decomposed tensor, and these regularities can sometimes speed up the computations. Furthermore, finding the regularities is a natural data analysis task. In this section, we will first give the definition of (Boolean) tensor CP decomposition and (Boolean) tensor rank. We will then study the properties of the decompositions, especially the sparsity. Unlike for the normal decomposition, with Boolean decompositions, we can prove upper bounds on the density of the factor matrices, showing that storing the data in a decomposed format is a valid option for saving space.

6.1 Definitions

The so-called *Boolean tensor CP decomposition*[3] is defined as follows:

Definition 1 ([19])**.** Given an n-by-m-by-l binary tensor $\boldsymbol{\mathcal{T}}$ and an integer r, its *rank-k Boolean CP decomposition* consists of three binary matrices $\boldsymbol{A}$ (n-by-r), $\boldsymbol{B}$ (m-by-r), and $\boldsymbol{C}$ (l-by-r) such that

$$\boldsymbol{\mathcal{T}} = \bigvee_{i=1}^{r} \boldsymbol{a}_{:i} \boxtimes \boldsymbol{b}_{:i} \boxtimes \boldsymbol{c}_{:i} \ . \tag{23}$$

The Boolean CP decomposition is closely connected to the concept of *Boolean tensor rank*.

Definition 2. A 3-way binary tensor $\boldsymbol{\mathcal{T}}$ is *rank-1 tensor* if $\boldsymbol{\mathcal{T}}$ is an outer product of three binary vectors, that is

$$\boldsymbol{\mathcal{T}} = \boldsymbol{a} \boxtimes \boldsymbol{b} \boxtimes \boldsymbol{c} \ . \tag{24}$$

The *Boolean rank* of a 3-way binary tensor $\boldsymbol{\mathcal{T}}$, denoted $\operatorname{rank}_B(\boldsymbol{\mathcal{T}})$, is the least integer r such that there exist r rank-1 binary tensors with

$$\boldsymbol{\mathcal{T}} = \bigvee_{i=1}^{r} \boldsymbol{a}_{:i} \boxtimes \boldsymbol{b}_{:i} \boxtimes \boldsymbol{c}_{:i} \ . \tag{25}$$

Notice that the ith columns of the *factor matrices* $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ of the CP decomposition define a rank-1 tensor $\boldsymbol{a}_{:i} \boxtimes \boldsymbol{b}_{:i} \boxtimes \boldsymbol{c}_{:i}$. In other words, the CP decomposition expresses the given tensor as a Boolean sum of r rank-1 tensors.

The Boolean CP decomposition can also be expressed in terms of Boolean and Khatri–Rao matrix products using matricization: three binary matrices $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ form a Boolean CP decomposition of $\boldsymbol{\mathcal{T}}$ if and only if [19]

$$\boldsymbol{T}_{(1)} = \boldsymbol{A} \circ (\boldsymbol{C} \odot \boldsymbol{B})^T \ , \tag{26}$$

$$\boldsymbol{T}_{(2)} = \boldsymbol{B} \circ (\boldsymbol{C} \odot \boldsymbol{A})^T \ , \tag{27}$$

and

$$\boldsymbol{T}_{(3)} = \boldsymbol{C} \circ (\boldsymbol{B} \odot \boldsymbol{A})^T \ . \tag{28}$$

Intuitively, then, we can think of the operation that turns the three factor matrices into a tensor as an operation that takes a join of two slices followed by distinct join of the result and another slice.

6.2 Properties of the CP Decomposition

Unfortunately, computing the rank or CP decomposition of a tensor is NP-hard, both under the normal [14] and Boolean [19] algebras. The rank is also not bounded by the smallest dimension, unlike with matrices. That is, there exist n-by-m-by-l binary tensors $\boldsymbol{\mathcal{T}}$ such that $\operatorname{rank}_B(\boldsymbol{\mathcal{T}}) > \min\{n, m, l\}$ [19]. However, we have an upper bound [19],

$$\operatorname{rank}_B(\boldsymbol{\mathcal{T}}) \leq \min\{nm, nl, ml\} \ . \tag{29}$$

Given the above results, we cannot hope for an efficient algorithm finding the smallest CP decomposition of data. Yet, we can always find *some* CP decomposition. The most naïve way is to unfold the data along the longest mode (so that the unfolded matrix has $\min\{nm, nl, ml\}$ columns), set this as one factor matrix, and then construct the other two factor matrices in such a way that their Khatri–Rao product is the identity matrix (how that is done is explained in [19]). Henceforth we will assume that our data tensor is represented in the factorized format.

A common motivation for storing data in factorized formats is that they can save space compared to storing the full data, essentially by a more efficient representation of regularities in the data. Yet, there usually are no studies on how much space the decomposition could save, or take. Boolean decompositions, however, do allow us to bound the density (or sparsity) of the factors with respect to the density of the data. First we repeat the result on the absolute number of non-zeroes in the factor matrices, from [19].

[3]The name is short for two names given to the same decomposition: CANDECOMP [8] and PARAFAC [13].

Proposition 6.1. Let $\boldsymbol{\mathcal{T}}$ be binary a tensor that has $\text{rank}_B(\boldsymbol{\mathcal{T}}) = r$. Then $\boldsymbol{\mathcal{T}}$ has a rank-r Boolean CP decomposition to $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ such that

$$|\boldsymbol{A}| + |\boldsymbol{B}| + |\boldsymbol{C}| \leq 3\,|\boldsymbol{\mathcal{T}}| \ . \tag{30}$$

Inequality (30) is tight (consider the case of $|\boldsymbol{\mathcal{T}}| = 1$), but it might paint a slightly too pessimistic picture of the actual sparsity. Rather, we would like to follow [11] and relate the sparsity of the factors to the sparsity of the data. If $\boldsymbol{\mathcal{T}}$ is an n-by-m-by-l binary tensor, we define its *sparsity* $s(\boldsymbol{\mathcal{T}})$ as

$$s(\boldsymbol{\mathcal{T}}) = 1 - \frac{|\boldsymbol{\mathcal{T}}|}{nml} \ . \tag{31}$$

(The same notation is extended to matrices and vectors with one (respectively two) modes having dimension 1.) Further, the rank-r Boolean CP decomposition $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ is *reducible* if there exists an index $j \in \{1, \ldots, r\}$ such that

$$\bigvee_{i=1,\ldots,r} \boldsymbol{a}_{:i} \boxtimes \boldsymbol{b}_{:i} \boxtimes \boldsymbol{c}_{:i} = \bigvee_{\substack{i=1,\ldots,r \\ i \neq j}} \boldsymbol{a}_{:i} \boxtimes \boldsymbol{b}_{:i} \boxtimes \boldsymbol{c}_{:i} \ . \tag{32}$$

If the decomposition is not reducible, it is said to be *irreducible*.

Proposition 6.2. Let $\boldsymbol{\mathcal{T}}$ be binary tensor that has a rank-r irreducible (but not necessarily minimal) Boolean CP decomposition to factors $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$. Then

$$s(\boldsymbol{A}) + s(\boldsymbol{B}) + s(\boldsymbol{C}) \geq s(\boldsymbol{\mathcal{T}}) \ . \tag{33}$$

Our proof uses a similar technique as [11]. We will first proof the following special case.

Lemma 1. *Proposition 6.2 holds when* $r = 1$.

Proof. Tensor $\boldsymbol{\mathcal{T}}$ must be rank-1. Hence, $|\boldsymbol{\mathcal{T}}| = |\boldsymbol{A}|\,|\boldsymbol{B}|\,|\boldsymbol{C}|$, or equivalently, $1 - s(\boldsymbol{\mathcal{T}}) = (1 - s(\boldsymbol{A}))(1 - s(\boldsymbol{B}))(1 - s(\boldsymbol{C}))$. It follows that

$$\begin{aligned} s(\boldsymbol{\mathcal{T}}) &= s(\boldsymbol{A}) + s(\boldsymbol{B}) + s(\boldsymbol{C}) - s(\boldsymbol{A})s(\boldsymbol{B}) - s(\boldsymbol{A})s(\boldsymbol{C}) - s(\boldsymbol{B})s(\boldsymbol{C}) + s(\boldsymbol{A})s(\boldsymbol{B})s(\boldsymbol{C}) \\ &\leq s(\boldsymbol{A}) + s(\boldsymbol{B}) + s(\boldsymbol{C}) \ , \end{aligned}$$

which holds as $s(\cdot) \in [0, 1]$. □

Proof of Proposition 6.2. For this proof we need the concept of a *residual tensor* $\boldsymbol{\mathcal{R}}_k$. Let $\boldsymbol{\mathcal{R}}_1 = \boldsymbol{\mathcal{T}}$, and for $k = 2, \ldots, r$, let $\boldsymbol{\mathcal{R}}_k = \boldsymbol{\mathcal{R}}_{k-1} \wedge \neg\,(\boldsymbol{a}_{:k} \boxtimes \boldsymbol{b}_{:k} \boxtimes \boldsymbol{c}_{:k})$, where the AND and NOT are element-wise. Notice that, as the decomposition is irreducible, $s(\boldsymbol{\mathcal{R}}_k) > s(\boldsymbol{\mathcal{R}}_{k-1})$.

We now have for $k = 1, \ldots, r$

$$s(\boldsymbol{a}_{:k}) + s(\boldsymbol{b}_{:k}) + s(\boldsymbol{c}_{:k}) \geq s(\boldsymbol{a}_{:k} \boxtimes \boldsymbol{b}_{:k} \boxtimes \boldsymbol{c}_{:k}) \geq s(\boldsymbol{\mathcal{R}}_k) \geq s(\boldsymbol{\mathcal{T}})$$

The first inequality follows from Lemma 1, and the next ones from the fact that the Boolean semi-ring is anti-negative. The statement follows, as

$$s(\boldsymbol{A}) = \frac{1}{r} \sum_{k=1}^{r} s(\boldsymbol{a}_{:k}),$$

and similarly for $s(\boldsymbol{B})$ and $s(\boldsymbol{C})$. □

The above discussion strongly indicates that storing the data tensor in the factorized form can yield significant space savings, and in the worst case, should not increase the storage requirements too much. It seems reasonable to assume that the factorisation would also give benefits on the computations: after all, low-rank tensors should have more regular structure than high-rank tensors, and this regularity should help with the computations. Unfortunately, there does not seem to be much work towards that end. Bader and Kolda [6] study some operations on CP-decomposed tensors (under normal algebra), and while their results generally carry over to the Boolean algebra, the operations they study are not commonly seen in our framework.

7 Other Related Work

Tensors, as a way to represent multi-way relations, have been studied in the context of databases for a long time, although the term *tensor* is not commonly used. Instead, terms like Data Cube [12] are used to refer to the data (and the associated operations, and the framework).

It is also not new to use the binary tensor representation of RDF data to improve the processing of SPARQL queries. For example, Atre et al. [4] propose a technique to effectively process join queries using binary tensors (referred to as *Bit-cubes*). Subsequent work also extends the method to left outer joins [3].

Tensor decompositions have been applied to RDF data earlier. For example, Drumond et al. [9] and Nickel et al. [23] use the decompositions to predict missing or unobserved RDF triples, while Erdős et al. [10] use so-called Boolean Tucker3 decomposition to discover facts from "surface" (s, p, o) triples.

For relational algebra, the tensor relational model [15] is a framework that supports both relational algebraic operations for data manipulation and tensor algebraic operations for data analysis. Kim and Candan propose efficient ways to combine the costly tensor decomposition needed for data analysis together with join [15], normalization [16], and union operations [17] for data manipulation. For the join for instance, the authors use non-negative tensor decompositions on the components to be joined in order to approximate the decomposition after the operation.

Bakibayev et al. [7] propose factorised relational databases that use compact factorized representations of data to improve query performance and to reduce redundancy. The authors propose a specialized query engine to handle select-project-join queries on such data efficiently.

8 Future Work

In this paper we have presented a framework for SPARQL queries as Boolean tensor operations. We believe that our framework can help with the analysis the SPARQL queries, and with the development of new optimizations.

The Boolean tensor rank measures the complexity of the tensor: the higher the rank, the less regularities the tensor has. It seems viable to use the tensor rank as a parameter of the complexity, much the same way as, say, the treewidth is used. Yet, we are unaware of any research towards that direction.

Most SPARQL operations (especially the joins) can be expressed using the Khatri–Rao product. Unlike the normal matrix product, that has enjoyed on significant research interest over the years, the Khatri–Rao product is relatively unknown and unstudied. As it is the key for evaluating joins, insights on the computation of it can lead to direct real-world benefits.

One can also ask the question the other way around, though. As the Khatri–Rao product (and the Boolean matrix product) can be expressed as SPARQL queries, could this be used as a way to implement more efficient algorithms for (Boolean) tensor analysis. A relatively simple goal could be to use the data structures and indexing approaches employed by RDF databases – such as RDF-3X [22] – for more efficient data analysis algorithms.

Going further, one can also consider implementing the whole data analysis algorithms on top of RDF databases. Again, we argue that our tensor representation should help, giving a framework where many data analysis problems map easily. It is clear, though, that SPARQL should be extended with new operations, should one want to implement more complicated data analysis directly on it (much the same ways as standard data mining methods can be integrated to relational databases; see, e.g. [20, 24]).

9 Conclusions

We have presented a framework for RDF and SPARQL based on Boolean tensors. The framework allows us to cast SPARQL queries as different types of matrix products, and use this formalisation to gain new insights and apply previously-defined techniques for their processing. Particularly, we showed how to count (and estimate) the cardinality of various SPARQL join operations. We also briefly considered the Boolean CP decomposition as an option to find regularities from the data, and use these regularities for improved query processing and storage. Whether the decompositions would help on query processing is still wide open, but we did show two upper bounds for the space requirements of the decomposition. Overall, we see this work only as the first step, as we believe that the framework we presented here facilitates better analysis of RDF and SPARQL.

References

[1] R. Amossen, A. Campagna, and R. Pagh. Better size estimation for sparse matrix products. *Algorithmica*, 69(3):741–757, 2014.

[2] R. R. Amossen and R. Pagh. Faster join-projects and sparse matrix multiplications. In *ICDT '09*, pages 121–126, Mar. 2009.

[3] M. Atre. A technique for SPARQL OPTIONAL (left-outer-join) queries. *CoRR*, abs/1304.7, 2013.

[4] M. Atre, V. Chaoji, M. J. Zaki, and J. A. Hendler. Matrix "Bit" Loaded: A Scalable Lightweight Join Query Processor for RDF Data. In *WWW '10*, pages 41—-50, 2010.

[5] B. Bader and T. Kolda. Algorithm 862: MATLAB tensor classes for fast algorithm prototyping. *ACM Transactions on Mathematical Software*, 32(4):635–653, 2006.

[6] B. Bader and T. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, 2007.

[7] N. Bakibayev, D. Olteanu, and J. Závodnỳ. Fdb: A query engine for factorised relational databases. *Proceedings of the VLDB Endowment*, 5(11):1232–1243, 2012.

[8] J. D. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an N-way generalization of "Eckart-Young" decomposition. *Psychometrika*, 35(3):283–319, 1970.

[9] L. Drumond, S. Rendle, and L. Schmidt-Thieme. Predicting RDF triples in incomplete knowledge bases with tensor factorization. *SAC '12*, page 326, 2012.

[10] D. Erdős and P. Miettinen. Discovering Facts with Boolean Tensor Tucker Decomposition. In *CIKM '13*, pages 1569–1572, 2013.

[11] N. Gillis and F. Glineur. Using underapproximations for sparse nonnegative matrix factorization. *Pattern Recogn.*, 43(4):1676–1687, 2010.

[12] J. Gray, S. Chaudhuri, and A. Bosworth. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *ICDE*, pages 152–159, 1996.

[13] R. A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an "explanatory" multimodal factor analysis. Technical Report 16, UCLA Working Papers in Phonetics, 1970.

[14] J. Håstad. Tensor rank is NP-complete. *J. Algorithm*, 11(4):644–654, Dec. 1990.

[15] M. Kim and K. Candan. Approximate tensor decomposition within a tensor-relational algebraic framework. *CIKM '11*, pages 1737–1742, 2011.

[16] M. Kim and K. Candan. Decomposition-by-normalization (DBN): leveraging approximate functional dependencies for efficient tensor decomposition. *CIKM '12*, pages 355–364, 2012.

[17] M. Kim and K. Candan. Pushing-down tensor decompositions over unions to promote reuse of materialized decompositions. *ECML PKDD '14*, pages 688–704, 2014.

[18] A. Lingas. A fast output-sensitive algorithm for boolean matrix multiplication. *Algorithmica*, pages 408–419, 2011.

[19] P. Miettinen. Boolean Tensor Factorizations. In *ICDM '11*, pages 447–456, 2011.

[20] A. Netz, S. Chaudhuri, U. Fayyad, and J. Bernhardt. Integrating data mining with SQL databases: OLE DB for data mining. In *ICDE '01*, pages 379–387, 2001.

[21] T. Neumann and G. Moerkotte. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *ICDE '11*, pages 984–994, 2011.

[22] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, Sept. 2009.

[23] M. Nickel, V. Tresp, and H.-P. Kriegel. Factorizing YAGO: Scalable Machine Learning for Linked Data. In *WWW '12*, pages 271–280, 2012.

[24] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating Association Rule Mining with Relational Database Systems: Alternatives and Implications. *Data Min. Knowl. Discov.*, 4(2-3):89–125, July 2000.