GigaTensor: Scaling Tensor Analysis Up By 100 Times - Algorithms and Discoveries

U Kang, Evangelos Papalexakis, Abhay Harpale, Christos Faloutsos
Carnegie Mellon University
{ukang, epapalex, aharpale, christos}@cs.cmu.edu

ABSTRACT

Many data are modeled as tensors, or multi dimensional arrays. Examples include the predicates (subject, verb, object) in knowledge bases, hyperlinks and anchor texts in the Web graphs, sensor streams (time, location, and type), social networks over time, and DBLP conference-author-keyword relations. Tensor decomposition is an important data mining tool with various applications including clustering, trend detection, and anomaly detection. However, current tensor decomposition algorithms are not scalable for large tensors with billions of sizes and hundreds millions of nonzeros: the largest tensor in the literature remains thousands of sizes and hundreds thousands of nonzeros.

Consider a knowledge base tensor consisting of about 26 million noun-phrases. The intermediate data explosion problem, associated with naive implementations of tensor decomposition algorithms, would require the materialization and the storage of a matrix whose largest dimension would be $\approx 7 \cdot 10^{14}$; this amounts to ~ 10 Petabytes, or equivalently a few data centers worth of storage, thereby rendering the tensor analysis of this knowledge base, in the naive way, practically impossible. In this paper, we propose GIGATENSOR, a scalable distributed algorithm for large scale tensor decomposition. GIGATENSOR exploits the sparseness of the real world tensors, and avoids the intermediate data explosion problem by carefully redesigning the tensor decomposition algorithm.

Extensive experiments show that our proposed GIGATENSOR solves $100\times$ bigger problems than existing methods. Furthermore, we employ GIGATENSOR in order to analyze a very large real world, knowledge base tensor and present our astounding findings which include discovery of potential synonyms among millions of noun-phrases (e.g. the noun 'pollutant' and the noun-phrase 'greenhouse gases').

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Design, Experimentation

Keywords

Tensor, Distributed Computing, Big Data, MapReduce, Hadoop

1. INTRODUCTION

Tensors, or multi-dimensional arrays appear in numerous applications: predicates (subject, verb, object) in knowledge bases [9], hyperlinks and anchor texts in the Web graphs [20], sensor streams (time, location, and type) [30], and DBLP conference-author-keyword relations [22], to name a few. Analysis of multi-dimensional arrays by tensor decompositions, as shown in Figure 1, is a basis for many interesting applications including clustering, trend detection, anomaly detection [22], correlation analysis [30], network forensic [25], and latent concept discovery [20].

There exist two, widely used, toolboxes that handle tensors and tensor decompositions: the Tensor Toolbox for Matlab [6], and the N-way Toolbox for Matlab [3]. Both toolboxes are considered the state of the art; especially, the Tensor Toolbox is probably the fastest existing implementation of tensor decompositions for sparse tensors (having attracted best paper awards, e.g. see [22]). However, the toolboxes have critical restrictions: 1) they operate strictly on data that can fit in the main memory, and 2) their scalability is limited by the scalability of Matlab. In [4, 22], efficient ways of computing tensor decompositions, when the tensor is very sparse, are introduced and are implemented in the Tensor Toolbox. However, these methods still operate in main memory and therefore cannot scale to Gigabytes or Terabytes of tensor data. The need for large scale tensor computations is ever increasing, and there is a huge gap that needs to be filled. In Table 1, we present an indicative sample of tensor sizes that have been analyzed so far; we can see that these sizes are nowhere near as adequate as needed, in order to satisfy current real data needs, which call for tensors with billions of sizes and hundreds of millions of nonzero elements.

In this paper, we propose GIGATENSOR, a scalable distributed algorithm for large scale tensor decomposition. GIGATENSOR can handle Tera-scale tensors using the MAPREDUCE [11] framework, and more specifically its open source implementation, HADOOP [1]. To the best of our knowledge, this paper is the first approach of deploying tensor decompositions in the MAPREDUCE framework. The main contributions of this paper are the following.

- **Algorithm.** We propose GIGATENSOR, a large scale tensor decomposition algorithm on MAPREDUCE. GIGATENSOR is carefully designed to minimize the intermediate data size and the number of floating point operations.
- **Scalability.** GIGATENSOR decomposes $100\times$ larger tensors compared to existing methods, as shown in Figure 4. Furthermore, GIGATENSOR enjoys near linear scalability on the number of machines.

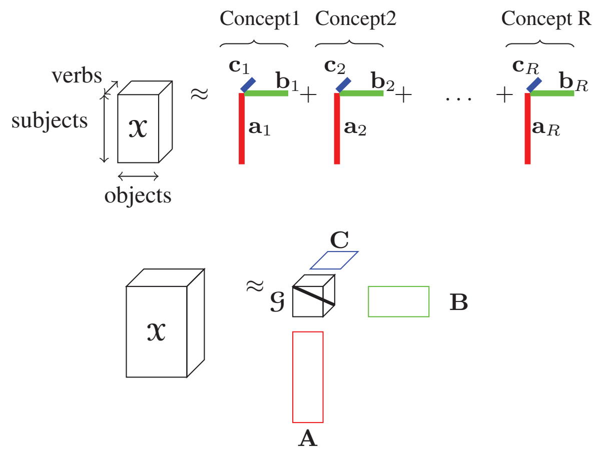


Figure 1: PARAFAC decomposition of three-way tensor as sum of R outer products (rank-one tensors), reminiscing of the rank-R SVD of a matrix (top), and as product of matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and a super-diagonal core tensor $\mathcal{G}$ (bottom).

Work	Tensor Size	Non-zeros
Kolda et al. [20]	787 × 787 × 533	3583
Acar et al. [2]	3.4 K × 100 × 18	(dense)
Maruhashi et al. [25]	2 K × 2 K × 6 K × 4 K	281 K
GIGATENSOR (**This work**)	26 M × 26 M × 48 M	144 M

Table 1: Indicative sizes of tensors analyzed in the data mining literature. Our proposed GIGATENSOR analyzes tensors with $\approx$ 1000× larger sizes and $\approx$ 500× larger nonzero elements.

- **Discovery.** We discover patterns in a very large knowledge-base tensor dataset from the 'Read the Web' project [9], which until now, was unable to be analyzed using tensor tools. Our findings include potential synonyms of noun-phrases, which were discovered after decomposing the knowledge base tensor; these findings are shown in Table 2 and a detailed description of the discovery procedure is covered on Section 4.

The rest of paper is organized as follows. Section 2 presents the preliminaries of the tensor decomposition. Sections 3 describes our proposed algorithm for large scale tensor analysis. Section 4 presents the experimental results. After reviewing related works in Section 5, we conclude in Section 6.

2. PRELIMINARIES; TENSOR DECOMPOSITION

In this section, we describe the preliminaries on the tensor decomposition whose fast algorithm will be proposed in Section 3. Table 3 lists the symbols used in this paper. For vector/matrix/tensor indexing, we use the Matlab-like notation: $\mathbf{A}(i,j)$ denotes the (i,j)-th element of matrix $\mathbf{A}$, whereas $\mathbf{A}(:,j)$ spans the j-th column of that matrix.

Matrices and the bilinear decomposition. Let $\mathbf{X}$ be an $I \times J$ matrix. The rank of $\mathbf{X}$ is the minimum number of rank one matrices that are required to compose $\mathbf{X}$. A rank one matrix is simply an *outer product* of two vectors, say $\mathbf{ab}^T$, where $\mathbf{a}$ and $\mathbf{b}$ are vectors. The (i,j)-th element of $\mathbf{ab}^T$ is simply $\mathbf{a}(i)\mathbf{b}(j)$. If rank($\mathbf{X}$) = R,

(Given) Noun Phrase	**(Discovered) Potential Contextual Synonyms**
pollutants	dioxin, sulfur dioxide, greenhouse gases, particulates, nitrogen oxide, air pollutants, cholesterol
disabilities	infections, dizziness, injuries, diseases, drowsiness, stiffness, injuries
vodafone	verizon, comcast
Christian history	European history, American history, Islamic history, history
disbelief	dismay, disgust, astonishment
cyberpunk	online-gaming
soul	body

Table 2: (Left:) Given noun-phrases; (right:) their potential *contextual* synonyms (i.e., terms with similar roles). They were automatically discovered using tensor decomposition of the NELL-1 knowledge base dataset (see Section 4 for details).

Symbol	Definition
$\mathcal{X}$	a tensor
$\mathbf{X}_{(\mathbf{n})}$	mode-n matricization of a tensor
m	number of nonzero elements in a tensor
a	a scalar (lowercase, italic letter)
$\mathbf{a}$	a column vector (lowercase, bold letter)
$\mathbf{A}$	a matrix (uppercase, bold letter)
R	number of components
$\circ$	outer product
$\odot$	Khatri-Rao product
$\otimes$	Kronecker product
$*$	Hadamard product
$\cdot$	standard product
$\mathbf{A}^T$	transpose of $\mathbf{A}$
$\mathbf{M}^\dagger$	pseudoinverse of $\mathbf{M}$
$\|\mathbf{M}\|_F$	Frobenius norm of $\mathbf{M}$
$bin(\mathbf{M})$	function that converts non-zero elements of $\mathbf{M}$ to 1

Table 3: Table of symbols.

then we can write

$$\mathbf{X} = \mathbf{a}_1\mathbf{b}_1^T + \mathbf{a}_2\mathbf{b}_2^T + \cdots + \mathbf{a}_R\mathbf{b}_R^T,$$

which is called the *bilinear decomposition* of $\mathbf{X}$. The bilinear decomposition is compactly written as $\mathbf{X} = \mathbf{AB}^T$, where the columns of $\mathbf{A}$ and $\mathbf{B}$ are $\mathbf{a}_r$ and $\mathbf{b}_r$, respectively, for $1 \le r \le R$. Usually, one may truncate this decomposition for $R \ll \text{rank}(\mathbf{X})$, in which case we have a low rank approximation of $\mathbf{X}$.

One of the most popular matrix decompositions is the Singular Value Decomposition (SVD):

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T,$$

where $\mathbf{U}, \mathbf{V}$ are unitary $I \times I$ and $J \times J$ matrices, respectively, and $\mathbf{\Sigma}$ is a rectangular diagonal matrix, containing the (non-negative) singular values of $\mathbf{X}$. If we pick $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{B} = \mathbf{V}$, and pick $R <$ rank($\mathbf{X}$), then we obtain the optimal low rank approximation of $\mathbf{X}$ in the least squares sense [13]. The SVD is also a very powerful tool used in computing the so called Moore-Penrose pseudoinverse [28], which lies in the heart of the PARAFAC tensor decomposition which we will describe soon. The Moore-Penrose pseudoinverse $\mathbf{X}^\dagger$ of $\mathbf{X}$ is given by

$$\mathbf{X}^\dagger = \mathbf{V}\mathbf{\Sigma}^{-1}\mathbf{U}^T.$$

We now provide a brief introduction to tensors and the PARAFAC decomposition. For a more detailed treatment of the subject, we refer the interested reader to [21].

Introduction to PARAFAC. Consider a three way tensor $\mathcal{X}$ of dimensions $I \times J \times K$; for the purposes of this initial analysis, we restrict ourselves to the study of three way tensors. The generalization to higher ways is trivial, provided that a robust implementation for three way decompositions exists.

DEFINITION 1 (THREE WAY OUTER PRODUCT). *The three way outer product of vectors* $\mathbf{a}, \mathbf{b}, \mathbf{c}$ *is defined as*

$$[\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}](i, j, k) = \mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k).$$

DEFINITION 2 (PARAFAC DECOMPOSITION). *The PARAFAC [14, 8] (also known as CP or trilinear) tensor decomposition of* $\mathcal{X}$ *in* R *components is*

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r.$$

The PARAFAC decomposition is a generalization of the matrix bilinear decomposition in three and higher ways. More compactly, we can write the PARAFAC decomposition as a triplet of matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, i.e. the r-th column of which contains $\mathbf{a}_r$, $\mathbf{b}_r$ and $\mathbf{c}_r$, respectively.

Furthermore, one may normalize each column of the three factor matrices, and introduce a scalar term λ_r, one for each rank-one factor of the decomposition (comprising a $R \times 1$ vector $\boldsymbol{\lambda}$), which forces the factor vectors to be of unit norm. Hence, the PARAFAC model we are computing is:

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r.$$

DEFINITION 3 (TENSOR UNFOLDING/MATRICIZATION). *We may unfold/matricize the tensor* $\mathcal{X}$ *in the following three ways:* $\mathbf{X_{(1)}}$ *of size* $(I \times JK)$, $\mathbf{X_{(2)}}$ *of size* $(J \times IK)$ *and* $\mathbf{X_{(3)}}$ *of size* $(K \times IJ)$. *The tensor* $\mathcal{X}$ *and the matricizations are mapped in the following way.*

$$\mathcal{X}(i, j, k) \rightarrow \mathbf{X_{(1)}}(i, j + (k-1)J). \quad (1)$$

$$\mathcal{X}(i, j, k) \rightarrow \mathbf{X_{(2)}}(j, i + (k-1)I). \quad (2)$$

$$\mathcal{X}(i, j, k) \rightarrow \mathbf{X_{(3)}}(k, i + (j-1)I). \quad (3)$$

We now set off to introduce some notions that play a key role in the computation of the PARAFAC decomposition.

DEFINITION 4 (KRONECKER PRODUCT). *The Kronecker product of* $\mathbf{A}$ *and* $\mathbf{B}$ *is:*

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} \mathbf{BA}(1,1) & \cdots & \mathbf{BA}(1, J_1) \\ \vdots & \ddots & \vdots \\ \mathbf{BA}(I_1, 1) & \cdots & \mathbf{BA}(I_1, J_1) \end{bmatrix}$$

If $\mathbf{A}$ *is of size* $I_1 \times J_1$ *and* $\mathbf{B}$ *of size* $I_2 \times J_2$, *then* $\mathbf{A} \otimes \mathbf{B}$ *is of size* $I_1 I_2 \times J_1 J_2$.

DEFINITION 5 (KHATRI-RAO PRODUCT). *The Khatri-Rao product (or column-wise Kronecker product)* $(\mathbf{A} \odot \mathbf{B})$, *where* $\mathbf{A}, \mathbf{B}$ *have the same number of columns, say* R, *is defined as:*

$$\mathbf{A} \odot \mathbf{B} = \left[\mathbf{A}(:,1) \otimes \mathbf{B}(:,1) \cdots \mathbf{A}(:,R) \otimes \mathbf{B}(:,R)\right]$$

If $\mathbf{A}$ *is of size* $I \times R$ *and* $\mathbf{B}$ *is of size* $J \times R$ *then* $(\mathbf{A} \odot \mathbf{B})$ *is of size* $IJ \times R$.

The Alternating Least Squares Algorithm for PARAFAC. The most popular algorithm for fitting the PARAFAC decomposition is the Alternating Least Squares (ALS). The ALS algorithm consists of three steps, each one being a conditional update of one of the three factor matrices, given the other two. The version of the algorithm we are using is the one outlined in Algorithm 1; for a detailed overview of the ALS algorithm, see [21, 14, 8].

Algorithm 1: Alternating Least Squares for the PARAFAC decomposition.

```
Input: Tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, rank $R$, maximum iterations $T$.
Output: PARAFAC decomposition $\boldsymbol{\lambda} \in \mathbb{R}^{R \times 1}$, $\mathbf{A} \in \mathbb{R}^{I \times R}$,
    $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$.
 1: Initialize $\mathbf{A}, \mathbf{B}, \mathbf{C}$;
 2: for $t = 1, ..., T$ do
 3:     $\mathbf{A} \leftarrow \mathbf{X_{(1)}} (\mathbf{C} \odot \mathbf{B}) (\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$;
 4:     Normalize columns of $\mathbf{A}$ (storing norms in vector $\boldsymbol{\lambda}$);
 5:     $\mathbf{B} \leftarrow \mathbf{X_{(2)}} (\mathbf{C} \odot \mathbf{A}) (\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^\dagger$;
 6:     Normalize columns of $\mathbf{B}$ (storing norms in vector $\boldsymbol{\lambda}$);
 7:     $\mathbf{C} \leftarrow \mathbf{X_{(3)}} (\mathbf{B} \odot \mathbf{A}) (\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^\dagger$;
 8:     Normalize columns of $\mathbf{C}$ (storing norms in vector $\boldsymbol{\lambda}$);
 9:     if convergence criterion is met then
10:         break for loop;
11:     end if
12: end for
13: return $\boldsymbol{\lambda}, \mathbf{A}, \mathbf{B}, \mathbf{C}$;
```

The stopping criterion for Algorithm 1 is either one of the following: 1) the maximum number of iterations is reached, or 2) the cost of the model for two consecutive iterations stops changing significantly (i.e. the difference between the two costs is within a small number ϵ, usually in the order of 10^{-6}). The cost of the model is simply the least squares cost.

The most important issue pertaining to the scalability of Algorithm 1 is the 'intermediate data explosion' problem. During the life of Algorithm 1, a naive implementation thereof will have to materialize matrices $(\mathbf{C} \odot \mathbf{B})$, $(\mathbf{C} \odot \mathbf{A})$, and $(\mathbf{B} \odot \mathbf{A})$, which are very large in sizes.

PROBLEM 1 (INTERMEDIATE DATA EXPLOSION). *The problem of having to materialize* $(\mathbf{C} \odot \mathbf{B})$, $(\mathbf{C} \odot \mathbf{A})$, $(\mathbf{B} \odot \mathbf{A})$ *is defined as the intermediate data explosion.*

In order to give an idea of how devastating this intermediate data explosion problem is, consider the NELL-1 knowledge base dataset, described in Section 4, that we are using in this work; this dataset consists of about $26 \cdot 10^6$ noun-phrases (and for a moment, ignore the number of the "context" phrases, which account for the third mode). Then, one of the intermediate matrices will have an explosive dimension of $\approx 7 \cdot 10^{14}$, or equivalently a few data centers worth of storage, rendering any practical way of materializing and storing it, virtually impossible.

In [4], Bader et al. introduce a way to alleviate the above problem, when the tensor is represented in a sparse form, in Matlab. This approach is however, as we mentioned earlier, bound by the memory limitations of Matlab. In Section 3, we describe our proposed method which effectively tackles intermediate data explosion, especially for sparse tensors, and is able to scale to very large tensors, because it operates on a distributed system.

3. PROPOSED METHOD

In this section, we describe GIGATENSOR, our proposed MAPREDUCE algorithm for large scale tensor analysis.

3.1 Overview

GIGATENSOR provides an efficient distributed algorithm for the PARAFAC tensor decomposition on MAPREDUCE. The major challenge is to design an efficient algorithm for updating factors (line 3, 5, and 7 of Algorithm 1). Since the update rules are similar, we focus on updating the $\mathbf{A}$ matrix. As shown in the line 3 of Algorithm 1, the update rule for A is

$$\hat{\mathbf{A}} \leftarrow \mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger, \tag{4}$$

where $\mathbf{X_{(1)}} \in \mathbb{R}^{I \times JK}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{JK \times R}$, and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \in \mathbb{R}^{R \times R}$. $\mathbf{X_{(1)}}$ is very sparse, especially in real world tensors, while $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are dense.

There are several challenges in designing an efficient MAPREDUCE algorithm for Equation (4) in GIGATENSOR:

1. **Minimize flops.** How to minimize the number of floating point operations (flops) for computing Equation (4)?
2. **Minimize intermediate data.** How to minimize the intermediate data size, i.e. the amount of network traffic in the shuffling stage of MAPREDUCE?
3. **Exploit data characteristics.** How to exploit the data characteristics including the sparsity of the real world tensor and the skewness in matrix multiplications to design an efficient MAPREDUCE algorithm?

We have the following main ideas to address the above challenges which we describe in detail in later subsections.

1. **Careful choice of order of computations** in order to minimize flops (Section 3.2).
2. **Avoiding intermediate data explosion** by exploiting the sparsity of real world tensors (Section 3.3 and 3.4.1).
3. **Parallel outer products** to minimize intermediate data (Section 3.4.2).
4. **Distributed cache multiplication** to minimize intermediate data by exploiting the skewness in matrix multiplications (Section 3.4.3).

3.2 Ordering of Computations

Equation (4) entails three matrix-matrix multiplications, assuming that we have already computed $(\mathbf{C} \odot \mathbf{B})$ and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$. Since matrix multiplication is commutative, Equation (4) can be computed by either multiplying the first two matrices, and multiplying the result with the third matrix:

$$[\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})](\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger, \tag{5}$$

or multiplying the last two matrices, and multiplying the first matrix with the result:

$$\mathbf{X_{(1)}}[(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger]. \tag{6}$$

The question is, which equation is better between (5) and (6)? From a standard result of numerical linear algebra (e.g. [7]), the Equation (5) requires $2mR + 2IR^2$ flops, where m is the number of nonzeros in the tensor $\mathcal{X}$, while the Equation (6) requires $2mR + 2JKR^2$ flops. Given that the product of the two dimension sizes (JK) is larger than the other dimension size (I) in most practical cases, Equation (5) results in smaller flops. For example, referring to the NELL-1 dataset of Table 7, Equation (5) requires $\approx 8 \cdot 10^9$ flops while Equation (6) requires $\approx 2.5 \cdot 10^{17}$ flops. For the reason, we choose the Equation (5) ordering for updating factor matrices. That is, we perform the following three matrix-matrix multiplications for Equation (4):

$$\texttt{Step 1:} \quad \mathbf{M}_1 \leftarrow \mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B}) \tag{7}$$

$$\texttt{Step 2:} \quad \mathbf{M}_2 \leftarrow (\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \tag{8}$$

$$\texttt{Step 3:} \quad \mathbf{M}_3 \leftarrow \mathbf{M}_1\mathbf{M}_2 \tag{9}$$

3.3 Avoiding the Intermediate Data Explosion Problem

As introduced at the end of Section 2, one of the most important issue for scaling up the tensor decomposition is the intermediate data explosion problem. In this subsection we describe the problem in detail, and propose our solution.

Problem. A naive algorithm to compute $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ is to first construct $\mathbf{C} \odot \mathbf{B}$, and multiply $\mathbf{X_{(1)}}$ with $\mathbf{C} \odot \mathbf{B}$, as illustrated in Figure 2. The problem ("intermediate data explosion ") of this algorithm is that although the matricized tensor $\mathbf{X_{(1)}}$ is sparse, the matrix $\mathbf{C} \odot \mathbf{B}$ is very large and dense; thus, $\mathbf{C} \odot \mathbf{B}$ cannot be stored even in multiple disks in a typical HADOOP cluster.

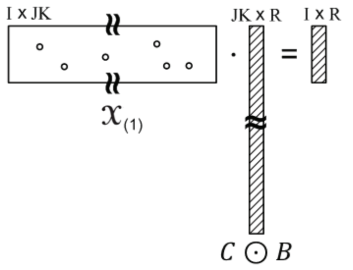


Figure 2: The "intermediate data explosion " problem in computing $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$. Although $\mathbf{X_{(1)}}$ is sparse, the matrix $\mathbf{C} \odot \mathbf{B}$ is very dense and long. Materializing $\mathbf{C} \odot \mathbf{B}$ requires too much storage: e.g., for $J = K \approx 26$ million as in the NELL-1 data of Table 7, $\mathbf{C} \odot \mathbf{B}$ explodes to *676 trillion* rows.

Our Solution. Our crucial observation is that $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ can be computed without explicitly constructing $\mathbf{C} \odot \mathbf{B}$. [1] Our main idea is to decouple the two terms in the Khatri-Rao product, and perform algebraic operations involving $\mathbf{X_{(1)}}$ and $\mathbf{C}$, then $\mathbf{X_{(1)}}$ and $\mathbf{B}$, and then combine the result. Our main idea is described in Algorithm 2 as well as in Figure 3. In line 7 of Algorithm 2, the Hadamard product of $\mathbf{X_{(1)}}$ and a matrix derived from $\mathbf{C}$ is performed. In line 8, the Hadamard product of $\mathbf{X_{(1)}}$ and a matrix derived from $\mathbf{B}$ is performed, where the $bin()$ function converts any nonzero value into 1, preserving sparsity. In line 9, the Hadamard product of the two result matrices from lines 7 and 8 is performed, and the elements of each row of the resulting matrix are summed up to get the final result vector $\mathbf{M}_1(:, r)$ in line 10. The following Theorem demonstrates the correctness of Algorithm 2.

THEOREM 1. *Computing* $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ *is equivalent to computing* $(\mathbf{N_1} * \mathbf{N_2}) \cdot \mathbf{1}_{JK}$*, where* $\mathbf{N}_1 = \mathbf{X_{(1)}} * (\mathbf{1}_I \circ (\mathbf{C}(:, r)^T \otimes \mathbf{1}_J^T))$*,* $\mathbf{N}_2 = bin(\mathbf{X_{(1)}}) * (\mathbf{1}_I \circ (\mathbf{1}_K^T \otimes \mathbf{B}(:, r)^T))$*, and* $\mathbf{1_{JK}}$ *is an all-1 vector of size* JK*.*

[1]Bader et al. [4] has an alternative way to avoid the intermediate data explosion.

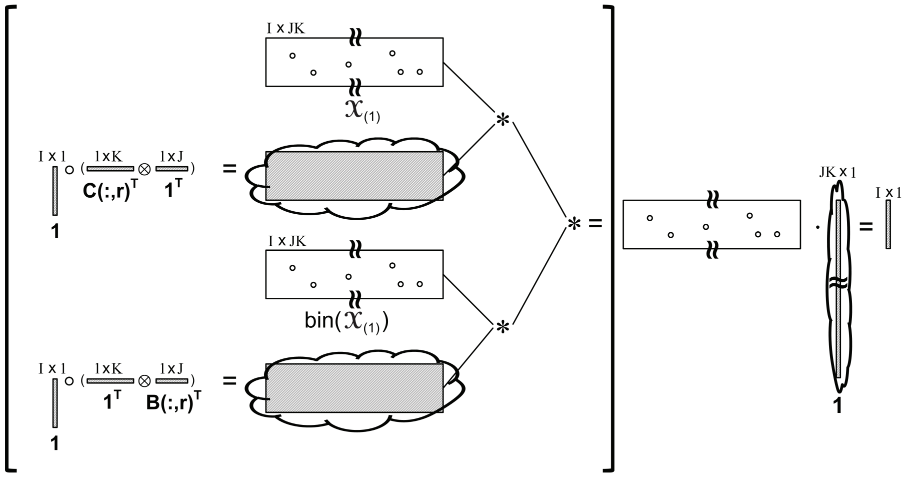


Figure 3: Our solution to avoid the intermediate data explosion. The main idea is to decouple the two terms in the Khatri-Rao product, and perform algebraic operations using $\mathbf{X}_{(1)}$ and $\mathbf{C}$, and then $\mathbf{X}_{(1)}$ with $\mathbf{B}$, and combine the result. The symbols $\circ, \otimes, *$, and $\cdot$ represents the outer, Kronecker, Hadamard, and the standard product, respectively. Shaded matrices are dense, and empty matrices with several circles are sparse. The clouds surrounding matrices represent that the matrices are *not* materialized. Note that the matrix $\mathbf{C} \odot \mathbf{B}$ is never constructed, and the largest dense matrix is either the $\mathbf{B}$ or the $\mathbf{C}$ matrix.

Algorithm 2: Multiplying $\mathbf{X}_{(1)}$ and $\mathbf{C} \odot \mathbf{B}$ in GIGATENSOR.

```
Input: Tensor $\mathbf{X}_{(1)} \in \mathbb{R}^{I \times JK}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$.
Output: $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$.
 1: $\mathbf{M_1} \leftarrow 0$;
 2: $\mathbf{1_I} \leftarrow$ all 1 vector of size $I$;
 3: $\mathbf{1_J} \leftarrow$ all 1 vector of size $J$;
 4: $\mathbf{1_K} \leftarrow$ all 1 vector of size $K$;
 5: $\mathbf{1_{JK}} \leftarrow$ all 1 vector of size $JK$;
 6: for $r = 1, ..., R$ do
 7:     $\mathbf{N}_1 \leftarrow \mathbf{X}_{(1)} * (\mathbf{1}_I \circ (\mathbf{C}(:, r)^T \otimes \mathbf{1}_J^T))$;
 8:     $\mathbf{N}_2 \leftarrow bin(\mathbf{X}_{(1)}) * (\mathbf{1}_I \circ (\mathbf{1}_K^T \otimes \mathbf{B}(:, r)^T))$;
 9:     $\mathbf{N}_3 \leftarrow \mathbf{N}_1 * \mathbf{N}_2$;
10:     $\mathbf{M}_1(:, r) \leftarrow \mathbf{N}_3 \cdot \mathbf{1}_{JK}$;
11: end for
12: return $\mathbf{M}_1$;
```

PROOF. The (i, y)-th element of $\mathbf{N}_1$ is given by

$$\mathbf{N}_1(i, y) = \mathbf{X}_{(1)}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r).$$

The (i, y)-th element of $\mathbf{N}_2$ is given by

$$\mathbf{N}_2(i, y) = \mathbf{B}(1 + (y - 1)\%J, r).$$

The (i, y)-th element of $\mathbf{N}_3 = \mathbf{N_1} * \mathbf{N_2}$ is

$$\mathbf{N}_3(i, y) = \mathbf{X}_{(1)}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r)\mathbf{B}(1 + (y - 1)\%J, r).$$

Multiplying $\mathbf{N}_3$ with $\mathbf{1}_{JK}$, which essentially sums up each row of $\mathbf{N}_3$, sets the i-th element $\mathbf{M}_1(i, r)$ of the $\mathbf{M}_1(:, r)$ vector equal to the following:

$$\mathbf{M}_1(i, r) = \sum_{y=1}^{JK} \mathbf{X}_{(1)}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r)\mathbf{B}(1 + (y - 1)\%J, r),$$

which is exactly the equation that we want from the definition of $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. $\square$

Notice that in Algorithm 2, the largest dense matrix required is either $\mathbf{B}$ or $\mathbf{C}$ (not $\mathbf{C} \odot \mathbf{B}$ as in the naive case), and therefore we have effectively avoided the intermediate data explosion problem.

Discussion. Table 4 compares the cost of the naive algorithm and GIGATENSOR for computing $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. The naive algorithm requires total $JKR+2mR$ flops (JKR for constructing $(\mathbf{C} \odot \mathbf{B})$, and $2mR$ for multiplying $\mathbf{X}_{(1)}$ and $(\mathbf{C} \odot \mathbf{B})$), and $JKR + m$ intermediate data size (JKR for $(\mathbf{C} \odot \mathbf{B})$, and m for $\mathbf{X}_{(1)}$). On the other hand, GIGATENSOR requires only $5mR$ flops ($3mR$ for three Hadamard products, and $2mR$ for the final multiplication), and $max(J + m, K + m)$ intermediate data size. The dependence on the term JK of the naive method makes it inappropriate for real world tensors which are sparse and the sizes of dimensions are much larger compared to the number m of nonzeros ($JK \gg m$). On the other hand, GIGATENSOR depends on $max(J+m, K+m)$ which is $O(m)$ for most practical cases, and thus fully exploits the sparsity of real world tensors.

Algorithm	**Flops**	**Intermediate Data**
Naive	$JKR + 2mR$	$JKR + m$
GIGATENSOR	$5mR$	$max(J + m, K + m)$

Table 4: Cost comparison of the naive and GIGATENSOR for computing $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$. J and K are the sizes of the second and the third dimensions, respectively, m is the number of nonzeros in the tensor, and R is the desired rank for the tensor decomposition (typically, $R \sim 10$). GIGATENSOR does not suffer from the intermediate data explosion problem, and is much more efficient than the naive algorithm in terms of both flops and intermediate data sizes. An arithmetic example, referring to the NELL-1 dataset of Table 7, for 8 bytes per value, and $R = 10$ is: $1.25 \cdot 10^{16}$ flops, 100 PB for the naive algorithm and $8.6 \cdot 10^9$ flops, 1.5GB for GIGATENSOR.

3.4 Our Optimizations for MapReduce

In this subsection, we describe MAPREDUCE algorithms for computing the three steps in Equations (7), (8), and (9).

3.4.1 Avoiding the Intermediate Data Explosion

The first step is to compute $\mathbf{M}_1 \leftarrow \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ (Equation (7)). The factors $\mathbf{C}$ and $\mathbf{B}$ are given in the form of $< j, r, \mathbf{C}(j,r) >$ and $< j, r, \mathbf{B}(j,r) >$, respectively. The tensor $\mathcal{X}$ is stored in the format of $< i, j, k, \mathcal{X}(i,j,k) >$, but we assume the tensor data is given in the form of mode-1 matricization ($< i, j, \mathbf{X}_{(1)}(i,j) >$) by using the mapping in Equation (1). We use Q_i and Q^j to denote the set of nonzero indices in $\mathbf{X}_{(1)}(i,:)$ and $\mathbf{X}_{(1)}(:,j)$, respectively: i.e., $Q_i = \{j | \mathbf{X}_{(1)}(i,j) > 0\}$ and $Q^j = \{i | \mathbf{X}_{(1)}(i,j) > 0\}$.

We first describe the MAPREDUCE algorithm for line 7 of Algorithm 2. Here, the tensor data and the factor data are joined for the Hadamard product. Notice that only the tensor $\mathcal{X}$ and the factor $\mathbf{C}$ are transferred in the shuffling stage.

- MAP-1: map $< i, j, \mathbf{X}_{(1)}(i,j) >$ on $\lceil \frac{j}{J} \rceil$, and $< j, r, \mathbf{C}(j,r) >$ on j such that tuples with the same key are shuffled to the same reducer in the form of $< j, (\mathbf{C}(j,r), \{(i, \mathbf{X}_{(1)}(i,j)) | \forall i \in Q^j\}) >$.
- REDUCE-1: take $< j, (\mathbf{C}(j,r), \{(i, X_{(1)}(i,j)) | \forall i \in Q^j\}) >$ and emit $< i, j, \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r) >$ for each $i \in Q^j$.

In the second MAPREDUCE algorithm for line 8 of Algorithm 2, we perform the similar task as the first MAPREDUCE algorithm but we do not multiply the value of the tensor, since line 8 uses the binary function. Again, only the tensor $\mathcal{X}$ and the factor $\mathbf{B}$ are transferred in the shuffling stage.

- MAP-2: map $< i, j, \mathbf{X}_{(1)}(i,j) >$ on $\lceil \frac{j}{J} \rceil$, and $< j, r, \mathbf{B}(j,r) >$ on j such that tuples with the same key are shuffled to the same reducer in the form of $< j, (\mathbf{B}(j,r), \{i | \forall i \in Q^j\}) >$.
- REDUCE-2: take $< j, (\mathbf{B}(j,r), \{i | \forall i \in Q^j\}) >$ and emit $< i, j, \mathbf{B}(j,r) >$ for each $i \in Q^j$.

Finally, in the third MAPREDUCE algorithm for lines 9 and 10, we combine the results from the first and the second steps using Hadamard product, and sums up each row to get the final result.

- MAP-3: map $< i, j, \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r) >$ and $< i, j, \mathbf{B}(j,r) >$ on i such that tuples with the same i are shuffled to the same reducer in the form of $< i, \{(j, \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r), \mathbf{B}(j,r))\} | \forall j \in Q_i >$.
- REDUCE-3: take $< i, \{(j, \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r), \mathbf{B}(j,r))\} | \forall j \in Q_i >$ and emit $< i, \sum_j \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r)\mathbf{B}(j,r) >$.

Note that the amount of data traffic in the shuffling stage is small (2 times the nonzeros of the tensor $\mathcal{X}$), considering that $\mathcal{X}$ is sparse.

3.4.2 Parallel Outer Products

The next step is to compute $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$ (Equation (8)). Here, the challenge is to compute $\mathbf{C^T C} * \mathbf{B^T B}$ efficiently, since once the $\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}$ is computed, the pseudo-inverse is trivial to compute because matrix $\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}$ is very small ($R \times R$ where R is very small; e.g. $R \sim 10$). The question is, how to compute $\mathbf{C^T C} * \mathbf{B^T B}$ efficiently? Our idea is to first compute $\mathbf{C}^T\mathbf{C}$, then $\mathbf{B}^T\mathbf{B}$, and performs the Hadamard product of the two $R \times R$ matrices. To compute $\mathbf{C}^T\mathbf{C}$, we express $\mathbf{C}^T\mathbf{C}$ as the sum of outer products of the rows:

$$\mathbf{C}^T\mathbf{C} = \sum_{k=1}^{K} \mathbf{C}(k,:)^T \circ \mathbf{C}(k,:), \tag{10}$$

where $\mathbf{C}(k,:)$ is the kth row of the $\mathbf{C}$ matrix. To implement the Equation (10) efficiently in MAPREDUCE, we partition the factor matrices row-wise [24]: we store each row of $\mathbf{C}$ into a line in the HADOOP File System (HDFS). The advantage of this approach compared to the column-wise partition is that each unit of data is self-joined with itself, and thus can be independently processed; column-wise partition would require each column to be joined with other columns which is prohibitively expensive.

The MAPREDUCE algorithm for Equation (10) is as follows.

- MAP: map $< j, \mathbf{C}(j,:) >$ on 0 so that all the output is shuffled to the only reducer in the form of $< 0, \{\mathbf{C}(j,:)^T \circ \mathbf{C}(j,:)\} \forall j >$.
- COMBINE, REDUCE: take $< 0, \{\mathbf{C}(j,:)^T \circ \mathbf{C}(j,:)\} \forall j >$ and emit $< 0, \sum_j \mathbf{C}(j,:)^T \circ \mathbf{C}(j,:) >$.

Since we use the combiner as well as the reducer, each mapper computes the local sum of the outer product. The result is that the size of the intermediate data, i.e. the number of input tuples to the reducer, is very small ($d \cdot R^2$ where d is the number of mappers) in GIGATENSOR. On the other hand, the naive column-wise partition method requires KR (the size of $\mathbf{C}^T$) + K (the size of a column of $\mathbf{C}$) intermediate data for 1 iteration, and thereby requires $K(R^2 + R)$ intermediate data for R iterations, which is much larger than the intermediate data size $d \cdot R^2$ of GIGATENSOR, as summarized in Table 5.

Algorithm	Flops	Intermediate Data	Example
Naive	KR^2	$K(R^2+R)$	40 GB
GIGATENSOR	KR^2	$d \cdot R^2$	40 KB

Table 5: Cost comparison of the naive (column-wise partition) method and GIGATENSOR for computing $\mathbf{C}^T\mathbf{C}$. K is the size of the third dimension, d is the number of mappers used, and R is the desired rank for the tensor decomposition (typically, $R \sim 10$). Notice that although the flops are the same for both methods, GIGATENSOR has much smaller intermediate data size compared to the naive method, considering $K \gg d$. The example refers to the intermediate data size for NELL-1 dataset of Table 7, for 8 bytes per value, $R = 10$ and $d = 50$.

3.4.3 Distributed Cache Multiplication

The final step is to multiply $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{I \times R}$ and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \in \mathbb{R}^{R \times R}$ (Equation (9)). We note the skewness of data sizes: the first matrix $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ is large and does not fit in the memory of a single machine, while the second matrix $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$ is very small to fit in the memory. To exploit this into better performance, we propose to use the distributed cache multiplication [16] to broadcast the second matrix to all the mappers that process the first matrix, and perform join in the first matrix. The result is that our method requires only one MAPREDUCE job with smaller intermediate data size (IR^2). On the other hand, the standard naive matrix-matrix multiplication requires two MAPREDUCE jobs: the first job for grouping the data by the column id of $\mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ and the row id of $(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})^\dagger$, and the second job for aggregation. Our proposed method is more efficient than the naive method since in the naive method the intermediate data size increases to $I(R+R^2)+R^2$ (the first job: $IR+R^2$, and the second job: IR^2), and the first job's output of size IR^2 should be written to and read from discs for the second job, as summarized in Table 6.

4. EXPERIMENTS

To evaluate our system, we perform experiments to answer the following questions:

Algorithm	Flops	Intermediate Data	Example
Naive	IR^2	$I(R+R^2)+R^2$	23 GB
GIGATENSOR	IR^2	IR^2	20 GB

Table 6: Cost comparison of the naive (column-wise partition) method and GIGATENSOR for multiplying $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$. I is the size of the first dimension, and R is the desired rank for the tensor decomposition (typically, $R \sim 10$). Although the flops are the same for both methods, GIGATENSOR has smaller intermediate data size compared to the naive method. The example refers to the intermediate data size for NELL-1 dataset of Table 7, for 8 bytes per value and $R = 10$.

Q1 What is the scalability of GIGATENSOR compared to other methods with regard to the sizes of tensors?
Q2 What is the scalability of GIGATENSOR compared to other methods with regard to the number of nonzero elements?
Q3 How does GIGATENSOR scale with regard to the number of machines?
Q4 What are the discoveries on real world tensors?

The tensor data in our experiments are summarized in Table 7, with the following details.

- NELL: real world knowledge base data containing (noun phrase 1, context, noun phrase 2) triples (e.g. 'George Harrison' 'plays' 'guitars') from the 'Read the Web' project [9]. NELL-1 data is the full data, and NELL-2 data is the filtered data from NELL-1 by removing entries whose values are below a threshold.
- Random: synthetic random tensor of size $I \times I \times I$. The size I varies from 10^4 to 10^9, and the number of nonzeros varies from 10^2 to $2 \cdot 10^7$.

Data	I	J	K	Nonzeros
NELL-1	26 M	26 M	48 M	144 M
NELL-2	15 K	15 K	29 K	77 M
Random	10 K∼1 B	10 K∼1 B	10 K∼1 B	100∼20 M

Table 7: Summary of the tensor data used. B: billion, M: million, K: thousand.

4.1 Scalability

We compare the scalability of GIGATENSOR and the Tensor Toolbox for Matlab [6] which is the current state of the art in terms of handling fast and effectively sparse tensors. The Tensor Toolbox is executed in a machine with a quad-core AMD 2.8 GHz CPU, 32 GB RAM, and 2.3 Terabytes disk. To run GIGATENSOR, we use CMU's OpenCloud HADOOP cluster where each machine has 2 quad-core Intel 2.83 GHz CPU, 16 GB RAM, and 4 Terabytes disk.

Scalability on the Size of Tensors. Figure 4 (a) shows the scalability of GIGATENSOR with regard to the sizes of tensors. We fix the number of nonzero elements to 10^4 on the synthetic data while increasing the tensor sizes $I = J = K$. We use 35 machines, and report the running time for 1 iteration of the algorithm. Notice that for smaller data the Tensor Toolbox runs faster than GIGATENSOR due to the overhead of running an algorithm on distributed systems, including reading/writing the data from/to disks, JVM loading type, and synchronization time. However as the data size grows beyond 10^7, the Tensor Toolbox runs out of memory while GIGATENSOR continues to run, eventually solving at least 100× larger problem than the competitor. We performed the same experiment while fixing the nonzero elements to 10^7, and we get similar results. We note that the Tensor Toolbox at its current implementation cannot run in distributed systems, thus we were unable to compare GIGATENSOR with a distributed Tensor Toolbox. We note that extending the Tensor Toolbox to run in a distributed setting is highly non-trivial; and even more complicated to make it handle data that don't fit in memory. On the contrary, our GIGATENSOR can handle such tensors.

Scalability on the Number of Nonzero Elements. Figure 4 (b) shows the scalability of GIGATENSOR compared to the Tensor Toolbox with regard to the number of nonzeros and tensor sizes on the synthetic data. We set the tensor size to be $I \times I \times I$, and the number of nonzero elements to be $I/50$. As in the previous experiment, we use 35 machines, and report the running time required for 1 iteration of the algorithm. Notice that GIGATENSOR decomposes tensors of sizes at least 10^9, while the Tensor Toolbox implementation runs out of memory on tensors of sizes beyond 10^7.

Scalability on the Number of Machines. Figure 5 shows the scalability of GIGATENSOR with regard to the number of machines. The Y-axis shows T_{25}/T_M where T_M is the running time for 1 iteration with M machines. Notice that the running time scales up near linearly.

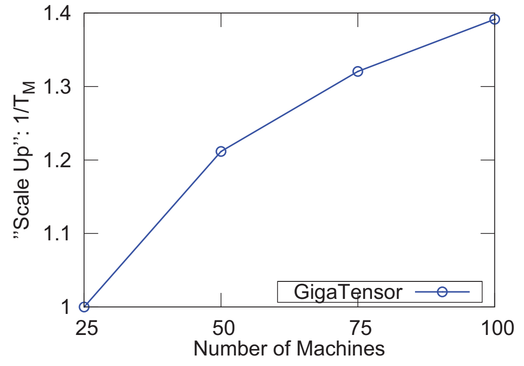
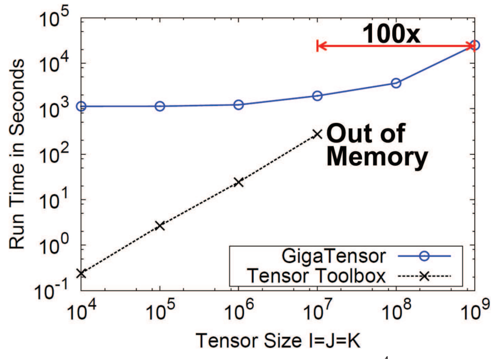


Figure 5: The scalability of GIGATENSOR with regard to the number of machines on the NELL-1 data. Notice that the running time scales up near linearly.

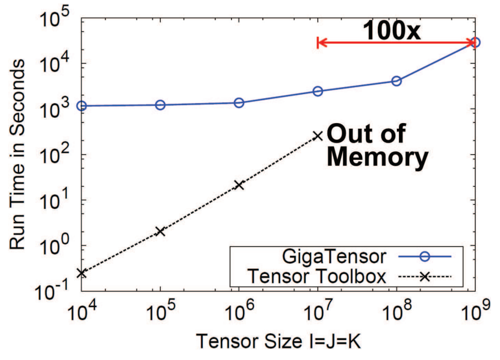
4.2 Discovery

In this section, we present discoveries on the NELL dataset that was previously introduced; we are mostly interested in demonstrating the power of our approach, as opposed to the current state of the art which was unable to handle a dataset of this magnitude. We perform two tasks: concept discovery, and (contextual) synonym detection.

Concept Discovery. With GIGATENSOR, we decompose the NELL-2 dataset in $R = 10$ components, and obtained $\lambda_i, \mathbf{A}, \mathbf{B}, \mathbf{C}$ (see Figure 1). Each one of the R columns of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ represents a grouping of similar (noun-phrase np1, noun-phrase np2, context words) triplets. The r-th column of $\mathbf{A}$ encodes with high values the noun-phrases in position np1, for the r-th group of triplets, the r-th column of $\mathbf{B}$ does so for the noun-phrases in position np2 and the r-th column of $\mathbf{C}$ contains the corresponding context words. In order to select the most representative noun-phrases and contexts for each group, we choose the k highest valued coefficients for each column. Table 8 shows 4 notable groups out of 10, and within each group the 3 most outstanding noun-phrases and contexts. Notice that each concept group contains relevant noun phrases and contexts.



(a) Number of nonzeros = 10^4.



(b) Number of nonzeros = $I/50$.

Figure 4: The scalability of GIGATENSOR compared to the Tensor Toolbox. (a) The number of nonzero elements is set to 10^4. (b) For a tensor of size $I \times I \times I$, the number of nonzero elements is set to $I/50$. In both cases, GIGATENSOR solves at least $100\times$ larger problem than the Tensor Toolbox which runs out of memory on tensors of sizes beyond 10^7.

Noun Phrase 1	Noun Phrase 2	Context
Concept 1: "Web Protocol"		
internet	protocol	'np1' 'stream' 'np2'
file	software	'np1' 'marketing' 'np2'
data	suite	'np1' 'dating' 'np2'
Concept 2: "Credit Cards"		
credit	information	'np1' 'card' 'np2'
Credit	debt	'np1' 'report' 'np2'
library	number	'np1' 'cards' 'np2'
Concept 3: "Health System"		
health	provider	'np1' 'care' 'np2'
child	providers	'np' 'insurance' 'np2'
home	system	'np1' 'service' 'np2'
Concept 4: "Family Life"		
life	rest	'np2' 'of' 'my' 'np1'
family	part	'np2' 'of' 'his' 'np1'
body	years	'np2' 'of' 'her' 'np1'

Table 8: Four notable groups that emerge from analyzing the NELL dataset.

Contextual Synonym Detection. The lower dimensional embedding of the noun phrases also permits a scalable and robust strategy for synonym detection. We are interested in discovering noun-phrases that occur in similar contexts, i.e. *contextual synonyms*. Using a similarity metric, such as Cosine Similarity, between the lower dimensional embeddings of the Noun-phrases (such as in the factor matrix $\mathbf{A}$), we can identify similar noun-phrases that can be used alternatively in sentence templates such as *np1 context np2*. Using the embeddings in the factor matrix $\mathbf{A}$ (appropriately column-weighted by λ), we get the synonyms that might be used in position *np1*, using $\mathbf{B}$ leads to synonyms for position *np2*, and using $\mathbf{C}$ leads to contexts that accept similar *np1* and *np2* arguments. In Table 2, which is located in Section 1, we show some exemplary synonyms for position *np1* that were discovered by this approach on NELL-1 dataset. Note that these are not synonyms in the traditional definition, but they are phrases that may occur in similar semantic roles in a sentence.

5. RELATED WORK

In this section, we review related works on tensor analysis (emphasizing on data mining applications), and MAPREDUCE/HADOOP.

Tensor Analysis. Tensors have a very long list of applications, pertaining to many fields additionally to data mining. For instance, tensors have been used extensively in Chemometrics [8] and Signal Processing [29]. Not very long ago, the data mining community has turned its attention to tensors and tensor decompositions. Some of the data mining applications that employ tensors are the following: in [20], Kolda et al. extend the famous HITS algorithm [19] by Kleinberg et al. in order to incorporate topical information in the links between the Web pages. In [2], Acar et al. analyze epilepsy data using tensor decompositions. In [5], Bader et al. employ tensors in order perform social network analysis, using the Enron dataset for evaluation. In [31], Sun et al. formulate click data on the Web pages as a tensor, in order to improve the Web search by incorporating user interests in the results. In [10], Chew et al. extend the Latent Semantic Indexing [12] paradigm for cross-language information retrieval, using tensors. In [33], Tao et al. employ tensors for 3D face modelling and in [32], a supervised learning framework, based on tensors is proposed. In [25], Maruhashi et al. present a framework for discovering bipartite graph like patterns in heterogeneous networks using tensors.

MAPREDUCE and HADOOP. MAPREDUCE is a distributed computing framework [11] for processing Web-scale data. MAPREDUCE has two advantages: (a) the data distribution, replication, fault-tolerance, and load balancing is handled automatically; and furthermore (b) it uses the familiar concept of functional programming. The programmer needs to define only two functions, a *map* and a *reduce*. The general framework is as follows [23]: (a) the *map* stage reads the input file and outputs (key, value) pairs; (b) the *shuffling* stage sorts the output and distributes them to reducers; (c) the *reduce* stage processes the values with the same key and outputs another (key, value) pairs which become the final result.

HADOOP [1] is the open source version of MAPREDUCE. HADOOP uses its own distributed file system HDFS, and provides a high-level language called PIG [26]. Due to its excellent scalability, ease of use, cost advantage, HADOOP has been used for many graph mining tasks (see [18, 15, 16, 17]).

6. CONCLUSION

In this paper, we propose GIGATENSOR, a tensor decomposition algorithm which scales to billion size tensors, and present interesting discoveries from real world tensors. Our major contributions include:

- **Algorithm.** We propose GIGATENSOR, a carefully designed large scale tensor decomposition algorithm on MAPREDUCE.
- **Scalability.** GIGATENSOR decomposes 100$\times$ larger tensors compared to previous methods, and GIGATENSOR scales near linearly to the number of machines.
- **Discovery.** We discover patterns of synonyms and concept groups in a very large knowledge base tensor which could not be analyzed before.

Future work could focus on related tensor decompositions, such as PARAFAC with sparse latent factors [27], as well as the TUCKER3 [34] decomposition.

Acknowledgement

Funding was provided by the U.S. ARO and DARPA under Contract Number W911NF-11-C-0088, by DTRA under contract No. HDTRA1-10-1-0120, and by ARL under Cooperative Agreement Number W911NF-09-2-0053, The views and conclusions are those of the authors and should not be interpreted as representing the official policies, of the U.S. Government, or other funding parties, and no official endorsement should be inferred. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

7. REFERENCES

[1] Hadoop information. http://hadoop.apache.org/.

[2] E. Acar, C. Aykut-Bingol, H. Bingol, R. Bro, and B. Yener. Multiway analysis of epilepsy tensors. *Bioinformatics*, 23(13):i10–i18, 2007.

[3] C.A. Andersson and R. Bro. The n-way toolbox for matlab. *Chemometrics and Intelligent Laboratory Systems*, 52(1):1–4, 2000.

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

[5] B.W. Bader, R.A. Harshman, and T.G. Kolda. Temporal analysis of social networks using three-way dedicom. *Sandia National Laboratories TR SAND2006-2161*, 2006.

[6] B.W. Bader and T.G. Kolda. Matlab tensor toolbox version 2.2. *Albuquerque, NM, USA: Sandia National Laboratories*, 2007.

[7] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[8] R. Bro. Parafac. tutorial and applications. *Chemometrics and intelligent laboratory systems*, 38(2):149–171, 1997.

[9] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010.

[10] P.A. Chew, B.W. Bader, T.G. Kolda, and A. Abdelali. Cross-language information retrieval using parafac2. In *KDD*, 2007.

[11] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[12] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, September 1990.

[13] C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, 1936.

[14] R.A. Harshman. Foundations of the parafac procedure: Models and conditions for an" explanatory" multimodal factor analysis. 1970.

[15] U. Kang, D. H. Chau, and C. Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *ICDE*, pages 243–254, 2011.

[16] U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD (2)*, pages 13–25, 2011.

[17] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, pages 1091–1099, 2011.

[18] U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system. In *ICDM*, pages 229–238, 2009.

[19] J.M. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM*, 46(5):604–632, 1999.

[20] T.G. Kolda and B.W. Bader. The tophits model for higher-order web link analysis. In *Workshop on Link Analysis, Counterterrorism and Security*, volume 7, pages 26–29, 2006.

[21] T.G. Kolda and B.W. Bader. Tensor decompositions and applications. *SIAM review*, 51(3), 2009.

[22] T.G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, 2008.

[23] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.

[24] C. Liu, H. c. Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690, 2010.

[25] K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *ASONAM*, 2011.

[26] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008.

[27] E.E. Papalexakis and N.D. Sidiropoulos. Co-clustering as multilinear decomposition with sparse latent factors. In *ICASSP*, 2011.

[28] R. Penrose. A generalized inverse for matrices. In *Proc. Cambridge Philos. Soc*, volume 51, pages 406–413. Cambridge Univ Press, 1955.

[29] N.D. Sidiropoulos, G.B. Giannakis, and R. Bro. Blind parafac receivers for ds-cdma systems. *Signal Processing, IEEE Transactions on*, 48(3):810–823, 2000.

[30] J. Sun, S. Papadimitriou, and P. S. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *ICDM*, pages 1076–1080, 2006.

[31] J.T. Sun, H.J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: a novel approach to personalized web search. In *WWW*, 2005.

[32] D. Tao, X. Li, X. Wu, W. Hu, and S.J. Maybank. Supervised tensor learning. *KAIS*, 13(1):1–42, 2007.

[33] D. Tao, M. Song, X. Li, J. Shen, J. Sun, X. Wu, C. Faloutsos, and S.J. Maybank. Bayesian tensor approach for 3-d face modeling. *IEEE TCSVT*, 18(10):1397–1410, 2008.

[34] L.R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3):279–311, 1966.