Fast and Scalable Distributed Boolean Tensor Factorization

Namyong Park
Seoul National University
Email: namyong.park@snu.ac.kr

Sejoon Oh
Seoul National University
Email: ohhenrie@snu.ac.kr

U Kang
Seoul National University
Email: ukang@snu.ac.kr

***Abstract*—How can we analyze tensors that are composed of 0's and 1's? How can we efficiently analyze such Boolean tensors with millions or even billions of entries? Boolean tensors often represent relationship, membership, or occurrences of events such as subject-relation-object tuples in knowledge base data (e.g., 'Seoul'-'is the capital of'-'South Korea'). Boolean tensor factorization (BTF) is a useful tool for analyzing binary tensors to discover latent factors from them. Furthermore, BTF is known to produce more interpretable and sparser results than normal factorization methods. Although several BTF algorithms exist, they do not scale up for large-scale Boolean tensors.**

In this paper, we propose DBTF, a distributed algorithm for Boolean tensor factorization running on the Spark framework. By caching computation results, exploiting the characteristics of Boolean operations, and with careful partitioning, DBTF successfully tackles the high computational costs and minimizes the intermediate data. Experimental results show that DBTF decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 68–382$\times$ less time, and exhibits near-linear scalability in terms of tensor dimensionality, density, rank, and machines.

I. Introduction

How can we analyze tensors that are composed of 0's and 1's? How can we efficiently analyze such Boolean tensors that have millions or even billions of entries? Many real-world data can be represented as tensors, or multi-dimensional arrays. Among them, many are composed of only either 0 or 1. Those tensors often represent relationship, membership, or occurrences of events. Examples of such data include subject-relation-object tuples in knowledge base data (e.g., 'Seoul'-'is the capital of'-'South Korea'), source IP-destination IP-port number-timestamp in network intrusion logs, and user1 ID-user2 ID-timestamp in friendship network data. Tensor factorizations are widely-used tools for analyzing tensors. CANDECOMP/PARAFAC (CP) and Tucker are two major tensor factorization methods [1]. These methods decompose a tensor into a sum of rank-1 tensors, from which we can find the latent structure of the data. Tensor factorization methods can be classified according to the constraint placed on the resulting rank-1 tensors [2]. The unconstrained form allows entries in the rank-1 tensors to be arbitrary real numbers, where we find linear relationships between latent factors; when a non-negativity constraint is imposed on the entries, the resulting factors reveal parts-of-whole relationships.

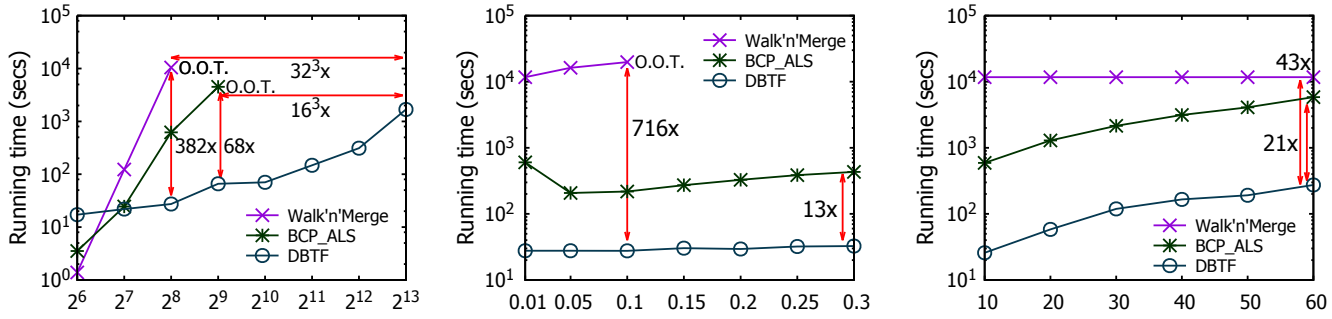
What we focus on in this paper is yet another approach with Boolean constraints, named Boolean tensor factorization (BTF) [3], that has many interesting applications including clustering, latent concept discovery, synonym finding, recommendation, and link prediction. BTF requires that the input tensor and all factor matrices be binary. Furthermore, BTF uses Boolean sum instead of normal addition, which means $1+1=1$ in BTF. When the data is binary, BTF is an appealing choice as it can reveal Boolean structures and relationships underlying the binary tensor that are hard to be found by other factorizations. Also, BTF is known to produce more interpretable and sparser results than the unconstrained and the non-negativity constrained counterparts, though at the expense of increased computational complexity [3], [4]. Several algorithms have been developed for BTF [3], [2], [5], [6]. While their scalability varies, they are not scalable enough for large-scale tensors with millions or even billions of non-zeros that have become widespread. The major challenges that need to be tackled for fast and scalable BTF are 1) how to minimize the computational costs involved with updating Boolean factor matrices, and 2) how to minimize the intermediate data that are generated in the process of factorization. Existing methods fail to solve both of these challenges.

TABLE I: Comparison of the scalability of our proposed DBTF and existing methods for Boolean tensor factorization. The scalability bottlenecks are colored red. As the only distributed approach, DBTF exhibits high scalability across all aspects of dimensionality, density, and rank; on the other hand, other methods show limited scalability for some aspects.

Method	Dimensionality	Density	Rank	Distributed
Walk'n'Merge [2]	Low	Low	High	No
BCP_ALS [3]	Low	High	High	No
DBTF	**High**	**High**	**High**	**Yes**

In this paper, we propose DBTF (Distributed Boolean Tensor Factorization), a distributed algorithm for Boolean CP factorization running on the Spark framework. DBTF tackles the high computational cost by utilizing caching in an efficient greedy algorithm for updating factor matrices, while minimizing the generation and shuffling of intermediate data. Also, DBTF exploits the characteristics of Boolean operations in solving both of the above problems. Due to the effective algorithm designed carefully with these ideas, DBTF achieves higher efficiency and scalability compared to existing methods. Table I shows a comparison of the scalability of DBTF and existing methods.

The main contributions of this paper are as follows:



(a) Dimensionality. (b) Density. (c) Rank.

Fig. 1: The scalability of DBTF and other methods with respect to the dimensionality, density, and rank of a tensor. o.o.t.: out of time (takes more than 6 hours). DBTF decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 68–382$\times$ less time (Figure 1(a)). Overall, DBTF achieves 13–716$\times$ speed-up, and exhibits near-linear scalability with regard to all data aspects.

- **Algorithm.** We propose DBTF, a distributed algorithm for Boolean CP factorization, which is designed to scale up to large tensors by minimizing intermediate data, caching computation results, and carefully partitioning the workload.
- **Theory.** We provide an analysis of the proposed algorithm in terms of time complexity, memory requirement, and the amount of shuffled data.
- **Experiment.** We present extensive empirical evidences for the scalability and performance of DBTF. The experimental results show that the proposed method decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 68–382$\times$ less time, as shown in Figure 1.

The binary code of our method and datasets used in this paper are available at http://datalab.snu.ac.kr/dbtf. The rest of the paper is organized as follows. Section II presents the preliminaries for the normal and Boolean CP factorizations. In Section III, we describe our proposed method for fast and scalable Boolean CP factorization. Section IV presents the experimental results. After reviewing related works in Section V, we conclude in Section VI.

II. Preliminaries

In this section, we present the notations and operations used for tensor decomposition, and define the normal and Boolean CP decompositions, After that, we introduce approaches for computing Boolean CP decomposition. Table II lists the definitions of symbols used in the paper.

A. Notation

We denote tensors by boldface Euler script letters (e.g., $\mathcal{X}$), matrices by boldface capitals (e.g., $\mathbf{A}$), vectors by boldface lowercase letters (e.g., $\mathbf{a}$), and scalars by lowercase letters (e.g., a).

Tensor. Tensor is a multi-dimensional array. The dimension of a tensor is also referred to as *mode* or *way*. A tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is an N-mode or N-way tensor. The $(i_1, i_2, \cdots, i_N)$-th entry of a tensor $\mathcal{X}$ is denoted by $x_{i_1 i_2 \cdots i_N}$. A colon in the subscript indicates taking all elements of that mode. For a three-way tensor $\mathcal{X}$, $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$ denote column (mode-1), row (mode-2), and tube (mode-3) fibers, respectively. $|\mathcal{X}|$ denotes the number of non-zero elements in a tensor $\mathcal{X}$; $\|\mathcal{X}\|$ denotes the Frobenius norm of a tensor $\mathcal{X}$, and is defined as $\sqrt{\sum_{i,j,k} x_{ijk}^2}$.

TABLE II: Table of symbols.

Symbol	Definition
$\mathcal{X}$	tensor (Euler script, bold letter)
$\mathbf{A}$	matrix (uppercase, bold letter)
$\mathbf{a}$	column vector (lowercase, bold letter)
a	scalar (lowercase, italic letter)
R	rank (number of components)
$\mathbf{X}_{(n)}$	mode-n matricization of a tensor $\mathcal{X}$
$\lvert\mathcal{X}\rvert$	number of non-zeros in the tensor $\mathcal{X}$
$\lVert\mathcal{X}\rVert$	Frobenius norm of the tensor $\mathcal{X}$
$\mathbf{A}^T$	transpose of matrix $\mathbf{A}$
$\circ$	outer product
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product
$\circledast$	pointwise vector-matrix product
$\mathbb{B}$	set of binary numbers, i.e., $\{0, 1\}$
$\vee$	Boolean sum of two binary tensors
$\bigvee$	Boolean summation of a sequence of binary tensors
$\boxtimes$	Boolean matrix product
I, J, K	dimensions of each mode of an input tensor $\mathcal{X}$

Tensor matricization/unfolding. The mode-n matricization (or unfolding) of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, denoted by $\mathbf{X}_{(n)}$, is the process of unfolding $\mathcal{X}$ into a matrix by rearranging its mode-n fibers to be the columns of the resulting matrix. For instance, a three-way tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and its matricizations are mapped as follows:

$$\begin{aligned} x_{ijk} &\rightarrow [\mathbf{X}_{(1)}]_{ic} \text{ where } c = j + (k-1)J \\ x_{ijk} &\rightarrow [\mathbf{X}_{(2)}]_{jc} \text{ where } c = i + (k-1)I \\ x_{ijk} &\rightarrow [\mathbf{X}_{(3)}]_{kc} \text{ where } c = i + (j-1)I. \end{aligned} \quad (1)$$

Outer product and rank-1 tensor. We use $\circ$ to denote the vector outer product. The three-way outer product of vectors $\mathbf{a} \in \mathbb{R}^I, \mathbf{b} \in \mathbb{R}^J$, and $\mathbf{c} \in \mathbb{R}^K$, is a tensor $\mathcal{X} = \mathbf{a} \circ \mathbf{b} \circ \mathbf{c} \in \mathbb{R}^{I \times J \times K}$ whose element (i, j, k) is defined as $(\mathbf{a} \circ \mathbf{b} \circ \mathbf{c})_{ijk} = a_i b_j c_k$. A three-way tensor $\mathcal{X}$ is rank-1 if it can be expressed as an outer product of three vectors.

Kronecker product. The Kronecker product of matrices $\mathbf{A} \in \mathbb{R}^{I_1 \times J_1}$ and $\mathbf{B} \in \mathbb{R}^{I_2 \times J_2}$ produces a matrix of size $I_1 I_2$-by-$J_1 J_2$, which is defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J_1}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J_1}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I_11}\mathbf{B} & a_{I_12}\mathbf{B} & \cdots & a_{I_1J_1}\mathbf{B} \end{bmatrix}. \quad (2)$$

Khatri-Rao product. The Khatri-Rao product (or column-wise Kronecker product) of matrices $\mathbf{A}$ and $\mathbf{B}$ that have the same number of columns, say R, is defined as:

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_{:1} \otimes \mathbf{b}_{:1} \ \mathbf{a}_{:2} \otimes \mathbf{b}_{:2} \ \cdots \ \mathbf{a}_{:R} \otimes \mathbf{b}_{:R}]. \quad (3)$$

If the sizes of $\mathbf{A}$ and $\mathbf{B}$ are I-by-R and J-by-R, respectively, that of $\mathbf{A} \odot \mathbf{B}$ is IJ-by-R.

Pointwise vector-matrix product. We define the pointwise vector-matrix product of a row vector $\mathbf{a} \in \mathbb{R}^R$ and a matrix $\mathbf{B} \in \mathbb{R}^{J \times R}$ as:

$$\mathbf{a} \circledast \mathbf{B} = [a_1\mathbf{b}_{:1} \ a_2\mathbf{b}_{:2} \ \cdots \ a_R\mathbf{b}_{:R}]. \quad (4)$$

Set of binary numbers. We use $\mathbb{B}$ to denote the set of binary numbers, that is, $\{0, 1\}$.

Boolean summation. We use $\bigvee$ to denote the Boolean summation, in which a sequence of Boolean tensors or matrices are summed. The Boolean sum ($\vee$) of two binary tensors $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and $\mathcal{Y} \in \mathbb{B}^{I \times J \times K}$, is defined by:

$$(\mathcal{X} \vee \mathcal{Y})_{ijk} = x_{ijk} \vee y_{ijk}. \quad (5)$$

The Boolean sum of two binary matrices is defined analogously.

Boolean matrix product. The Boolean product of two binary matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$ and $\mathbf{B} \in \mathbb{B}^{R \times J}$ is defined as:

$$(\mathbf{A} \boxtimes \mathbf{B})_{ij} = \bigvee_{k=1}^{R} a_{ik} b_{kj}. \quad (6)$$

B. Tensor Rank and Decomposition

1) Normal tensor rank and CP decomposition: With the above notations, we first define the normal tensor rank and CP decomposition.

Definition 1. *(Tensor rank)* The rank of a three-way tensor $\mathcal{X}$ is the smallest integer R such that there exist R rank-1 tensors whose sum is equal to the tensor $\mathcal{X}$, i.e.,

$$\mathcal{X} = \sum_{i=1}^{R} \boldsymbol{a}_i \circ \boldsymbol{b}_i \circ \boldsymbol{c}_i. \quad (7)$$

Definition 2. *(CP decomposition)* Given a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and a rank R, find factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$ such that they minimize

$$\left\| \mathcal{X} - \sum_{i=1}^{R} \boldsymbol{a}_i \circ \boldsymbol{b}_i \circ \boldsymbol{c}_i \right\|. \quad (8)$$

CP decomposition can be expressed in a matricized form as follows [1]:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T \\ \mathbf{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^T \\ \mathbf{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^T. \end{aligned} \quad (9)$$

2) Boolean tensor rank and CP decomposition: We now define the Boolean tensor rank and CP decomposition. The definitions of Boolean tensor rank and CP decomposition differ from their normal counterparts in the following two respects: 1) the tensor and factor matrices are binary; 2) Boolean sum is used where $1 + 1$ is defined to be 1.

Definition 3. *(Boolean tensor rank)* The Boolean rank of a three-way binary tensor $\mathcal{X}$ is the smallest integer R such that there exist R rank-1 binary tensors whose Boolean summation is equal to the tensor $\mathcal{X}$, i.e.,

$$\mathcal{X} = \bigvee_{i=1}^{R} \boldsymbol{a}_i \circ \boldsymbol{b}_i \circ \boldsymbol{c}_i. \quad (10)$$

Definition 4. *(Boolean CP decomposition)* Given a binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and a rank R, find binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$ such that they minimize

$$\left| \mathcal{X} - \bigvee_{i=1}^{R} \boldsymbol{a}_i \circ \boldsymbol{b}_i \circ \boldsymbol{c}_i \right|. \quad (11)$$

Boolean CP decomposition can be expressed in matricized form as follows [3]:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T \\ \mathbf{X}_{(2)} &\approx \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^T \\ \mathbf{X}_{(3)} &\approx \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^T. \end{aligned} \quad (12)$$

Figure 2 illustrates the rank-R Boolean CP decomposition of a three-way tensor.

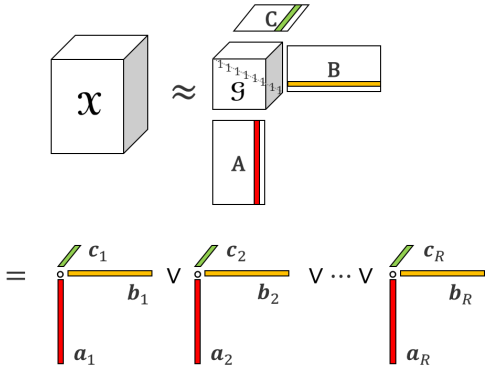


Fig. 2: Rank-R Boolean CP decomposition of a three-way tensor $\mathcal{X}$. $\mathcal{X}$ is decomposed into three Boolean factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$.

Computing the Boolean CP decomposition. The alternating least squares (ALS) algorithm is the "workhorse" approach for normal CP decomposition [1]. With a few changes, ALS projection heuristic provides a framework for computing the Boolean CP decomposition as shown in Algorithm 1.

The framework in Algorithm 1 is composed of two parts: first, the initialization of factor matrices (line 2), and second, the iterative update of each factor matrix in turn (lines 4-6). At each step of the iterative update phase, the n-th factor matrix is updated given the mode-n matricization of the input tensor $\mathcal{X}$ with the goal of minimizing the difference between the input

Algorithm 1: Boolean CP Decomposition Framework

Input: A three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, rank R, and maximum iterations T.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$.

1 initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
2 **for** $t \leftarrow 1..T$ **do**
3 update $\mathbf{A}$ such that $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T|$ is minimized
4 update $\mathbf{B}$ such that $|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^T|$ is minimized
5 update $\mathbf{C}$ such that $|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^T|$ is minimized
6 **if** *converged* **then**
7 break out of **for** loop
8 **return** $\mathbf{A}$, $\mathbf{B}$, *and* $\mathbf{C}$

tensor $\mathcal{X}$ and the approximate tensor reconstructed from the factor matrices, while the other factor matrices are fixed.

The convergence criterion for Algorithm 1 is either one of the following: 1) the number of iterations exceeds the maximum value T, or 2) the sum of absolute differences between the input tensor and the reconstructed one does not change significantly for two consecutive iterations (i.e., the difference between the two errors is within a small threshold).

Using the above framework, Miettinen [3] proposed a Boolean CP decomposition method named BCP_ALS. However, since BCP_ALS is designed to run on a single machine, the scalability and performance of BCP_ALS are limited by the computing and memory capacity of a single machine. Also, the initialization scheme used in BCP_ALS has high space and time requirements which are proportional to the squares of the number of columns of each unfolded tensor. Due to these limitations, BCP_ALS cannot scale up to large-scale tensors. Walk'n'Merge [2] is a different approach for Boolean tensor factorization: representing the tensor as a graph, Walk'n'Merge performs random walks on it to identify dense blocks (rank-1 tensors), and merge these blocks to get larger, yet dense blocks. While Walk'n'Merge is a parallel algorithm, its scalability is still limited. Since it is not a distributed method, Walk'n'Merge suffers from the same limitations of a single machine. Also, as the size of tensor increases, the running time of Walk'n'Merge rapidly increases as we show in Section IV-B.

III. Proposed Method

In this section, we describe DBTF, our proposed method for distributed Boolean tensor factorization. There are several challenges to efficiently perform Boolean tensor factorization in a distributed environment.

1) **Minimize intermediate data.** The amount of intermediate data that are generated and shuffled across machines affects the performance of a distributed algorithm significantly. How can we minimize the intermediate data?
2) **Minimize flops.** Boolean tensor factorization is an NP-hard problem [3] with a high computational cost. How can we minimize the number of floating point operations (flops) for updating factor matrices?
3) **Exploit the characteristics of Boolean operations.** In contrast to the normal tensor factorization, Boolean tensor factorization applies Boolean operations to binary data. How can we exploit the characteristics of Boolean operations to design an efficient and scalable algorithm?

We address the above challenges with the following main ideas, which we describe in later subsections.

1) **Distributed generation and minimal transfer of intermediate data** remove redundant data generation and reduce the amount of data transfer. (Section III-B).
2) **Caching intermediate computation results** decreases the number of flops remarkably by exploiting the characteristics of Boolean operations. (Section III-C).
3) **Careful partitioning of the workload** facilitates reuse of intermediate results and minimizes data shuffling. (Section III-D).

We first give an overview of how DBTF updates the factor matrices (Section III-A), and then describe how we address the aforementioned scalability challenges in detail (Sections III-B to III-D). After that, we give a theoretical analysis of DBTF (Section III-G).

A. Overview

DBTF is a distributed Boolean CP decomposition algorithm based on the framework described in Algorithm 1. The core operation of DBTF is updating the factor matrix (lines 3-5 in Algorithm 1). Since the update steps are similar, we focus on updating the factor matrix $\mathbf{A}$. DBTF performs a column-wise update row by row: DBTF iterates over the rows of factor matrix for R column (outer)-iterations in total, updating column c ($1 \leq c \leq R$) of each row at column-iteration c. Figure 3 shows an overview of how DBTF updates a factor matrix. In Figure 3, the red rectangle indicates the column c currently being updated, and the gray rectangle in $\mathbf{A}$ refers to the row DBTF is visiting in row (inner)-iteration i.

The objective of updating the factor matrix is to minimize the difference between $\mathbf{X}_{(1)}$ and $\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T$. To do so,

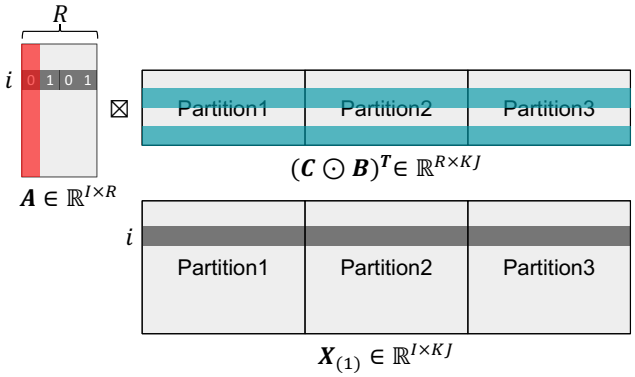


Fig. 3: An overview of updating a factor matrix. DBTF performs a column-wise update row by row: DBTF iterates over the rows of factor matrix for R column (outer)-iterations in total, updating column c ($1 \leq c \leq R$) of each row at column-iteration c. The red rectangle in $\mathbf{A}$ indicates the column c currently being updated; the gray rectangle in $\mathbf{A}$ refers to the row DBTF is visiting in row (inner)-iteration i; blue rectangles in $(\mathbf{C} \odot \mathbf{B})^T$ are the rows that are Boolean summed to be compared against the i-th row of $\mathbf{X}_{(1)}$ (gray rectangle in $\mathbf{X}_{(1)}$). Vertical blocks in $(\mathbf{C} \odot \mathbf{B})^T$ and $\mathbf{X}_{(1)}$ represent partitioning of the data (see Section III-D for details on partitioning).

DBTF computes $\left|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T\right|$ for combinations of values of entries in column c (i.e., $\mathbf{a}_{:c}$), and update column c to the set of values that yield the smallest difference. To calculate the difference at row-iteration i, DBTF compares $[\mathbf{X}_{(1)}]_{i:}$ (gray rectangle in $\mathbf{X}_{(1)}$ in Figure 3) against $[\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T]_{i:} = \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^T$. Then an entry a_{ic} is updated to the value that gives a smaller difference $\left|[\mathbf{X}_{(1)}]_{i:} - \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^T\right|$.

Lemma 1. *$\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^T$ is the same as selecting rows of $(\mathbf{C} \odot \mathbf{B})^T$ that correspond to the indices of non-zeros of $\mathbf{a}_{i:}$, and performing a Boolean summation of those rows.*

Proof. This follows from the definition of Boolean matrix product $\boxtimes$ (Equation 6). □

Consider Figure 3 as an example: since $\mathbf{a}_{i:}$ is 0101 (gray rectangle in $\mathbf{A}$), $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^T$ is identical to the Boolean summation of the second and fourth rows (blue rectangles).

Note that an update of the i-th row of $\mathbf{A}$ does not depend on those of other rows since $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^T$ needs to be compared only with $[\mathbf{X}_{(1)}]_{i:}$. Therefore, the determination of whether to update column entries in $\mathbf{A}$ to 0 or 1 can be made independently of each other.

B. Distributed Generation and Minimal Transfer of Intermediate Data

The first challenge for updating a factor matrix in a distributed manner is how to generate and distribute the intermediate data efficiently. Updating a factor matrix involves two types of intermediate data: 1) a Khatri-Rao product of two factor matrices (e.g., $(\mathbf{C} \odot \mathbf{B})^T$), and 2) an unfolded tensor (e.g., $\mathbf{X}_{(1)}$).

Khatri-Rao product. A naive method for processing the Khatri-Rao product is to construct the entire Khatri-Rao product first, and then distribute its partitions across machines. While Boolean factors are known to be sparser than the normal counterparts with real-valued entries [4], performing the entire Khatri-Rao product is still an expensive operation. Also, since one of the two matrices involved in the Khatri-Rao product is always updated in the previous update procedure (Algorithm 1), prior Khatri-Rao products cannot be reused. Our idea is to distribute only the factor matrices, and then let each machine generate the part of the product it needs, which is possible according to the definition of Khatri-Rao product:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{b}_{:1} & a_{12}\mathbf{b}_{:2} & \cdots & a_{1R}\mathbf{b}_{:R} \\ a_{21}\mathbf{b}_{:1} & a_{22}\mathbf{b}_{:2} & \cdots & a_{2R}\mathbf{b}_{:R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{b}_{:1} & a_{I2}\mathbf{b}_{:2} & \cdots & a_{IR}\mathbf{b}_{:R} \end{bmatrix}. \quad (13)$$

We notice from Equation (13) that a specific range of rows of Khatri-Rao product can be constructed if we have the two factor matrices and the corresponding range of row indices. With this change, we only need to broadcast relatively small factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ along with the index ranges assigned for each machine without having to materialize the entire Khatri-Rao product.

Unfolded Tensor. While the Khatri-Rao products are computed iteratively, matricizations of an input tensor need to be performed only once. However, in contrast to the Khatri-Rao product, we cannot avoid shuffling the entire unfolded tensor as we have no characteristics to exploit as in the case of Khatri-Rao product. Furthermore, unfolded tensors take up the largest space during the execution of DBTF. In particular, its row dimension quickly becomes very large as the sizes of factor matrices increase. Therefore, we partition the unfolded tensors in the beginning, and do not shuffle them afterwards. We do vertical partitioning of both the Khatri-Rao product and unfolded tensors as shown in Figure 3 (see Section III-D for more details on partitioning).

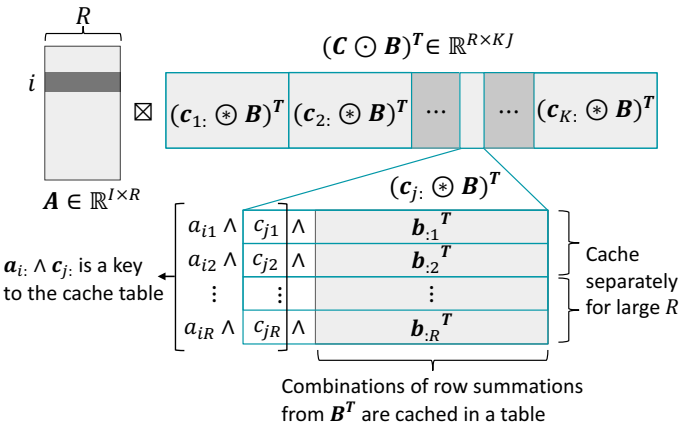


Fig. 4: An overview of caching. Blue rectangles in $(\mathbf{C} \odot \mathbf{B})^T$ correspond to K pointwise vector-matrix products, among which $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$ is shown in detail. $\mathbf{B}^T$ is a unit of caching: combinations of its row summations are cached in a table. $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ determines which rows are to be used for the row summation of $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$. For large R, rows of $\mathbf{B}^T$ are split into multiple, smaller groups, each of which is cached separately.

C. Caching of Intermediate Computation Results

The second and the most important challenge for efficient and scalable Boolean tensor factorization is how to minimize the number of floating point operations (flops) for updating factor matrices. In this subsection, we describe the problem in detail, and present our solution.

Problem. Given our procedure to update factor matrices (Section III-A), the two most frequent operations are 1) computing the Boolean sums of selected rows of $(\mathbf{C} \odot \mathbf{B})^T$, and 2) comparing the resulting row with the corresponding row of $\mathbf{X}_{(1)}$. Assuming that all factor matrices are of the same size, I-by-R, these operations take $O(RI^2)$ and $O(I^2)$ time, respectively. Since we compute the errors for both cases of when each factor matrix entry is set to 0 and 1, each operation needs to be performed $2RI$ times to update a factor matrix of size I-by-R; then, updating all three factor matrices for T iterations performs each operation $6TRI$ times in total. Due to the high computational costs and large number of repetitions, it is crucial to minimize the number of flops for these operations.

Our Solution. We start from the following observations:

- By Lemma 1, DBTF computes the Boolean sum of selected rows in $(\mathbf{C} \odot \mathbf{B})^T$. This amounts to performing a specific set of operations repeatedly, which we describe below.

- Given the rank R, the number of combinations of selecting rows in $(\mathbf{C} \odot \mathbf{B})^T$ is 2^R.

Our main idea is to precompute the set of operations that will be performed repeatedly, cache the results, and reuse them for all possible Boolean row summations. Figure 4 gives an overview of our idea for caching. We note that from the definitions of the Khatri-Rao (Equation (3)) and the pointwise vector-matrix product (Equation (4)),

$$(\mathbf{C} \odot \mathbf{B})^T = [(\mathbf{c}_{1:} \circledast \mathbf{B})^T \ (\mathbf{c}_{2:} \circledast \mathbf{B})^T \ \cdots \ (\mathbf{c}_{K:} \circledast \mathbf{B})^T].$$

In Figure 4, blue rectangles in $(\mathbf{C} \odot \mathbf{B})^T$ correspond to K pointwise vector-matrix (PVM) products. Since a row of $(\mathbf{C} \odot \mathbf{B})^T$ is made up of a sequence of K corresponding rows of PVM products $(\mathbf{c}_{1:} \circledast \mathbf{B})^T, \ldots, (\mathbf{c}_{K:} \circledast \mathbf{B})^T$, the Boolean sum of selected rows of $(\mathbf{C} \odot \mathbf{B})^T$ can be constructed by summing up the same set of rows in each PVM product, and concatenating the resulting rows into a single row.

Given that the row $\mathbf{a}_{i:}$ is being updated as in Figure 4, we notice that computing Boolean row summations of each $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$ amounts to summing up the rows in $\mathbf{B}^T$ that are selected by the next two conditions. First, we choose all those rows of $\mathbf{B}^T$ whose corresponding entries in $\mathbf{c}_{j:}$ are 1. Since all other rows are empty vectors by the definition of Khatri-Rao product, they can be ignored in computing Boolean row summations. Second, we pick the set of rows from each $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$ selected by the value of row $\mathbf{a}_{i:}$ as they are the targets of Boolean summation. Therefore, the value of Boolean AND ($\wedge$) between the rows $\mathbf{a}_{i:}$ and $\mathbf{c}_{j:}$ determines which rows are to be used for the row summation of $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$.

In computing a row summation of $(\mathbf{C} \odot \mathbf{B})^T$, we repeatedly sum a subset of rows in $\mathbf{B}^T$ selected by the above conditions for each PVM product. Then, if we have the results for all combinations of row summations of $\mathbf{B}^T$, we can avoid summing up the same set of rows over and over again. DBTF precalculates these combinations, and caches the results in a table in memory. This tables maps a specific subset of selected rows in $\mathbf{B}^T$ to their Boolean summation result; we use $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ as a key to this table.

An issue related with this approach is that the space required for the table increases exponentially with R. For example, when the rank R is 20, we need a table that can store $2^{20} \approx 1,000,000$ row summations. Since this is infeasible for large R, when R becomes larger than a threshold value V, we divide the rows evenly into $\lceil R/V \rceil$ smaller groups, construct smaller tables for each group, and then perform additional Boolean summation of rows that come from the smaller tables.

Lemma 2. *Given R and V, the number of required cache tables is $\lceil R/V \rceil$, and each table is of size $2^{\lceil R/\lceil R/V \rceil \rceil}$.*

For instance, when the rank R is 18 and V is set to 10, we create two tables of size 2^9, the first one storing possible summations of ${\mathbf{b}_{:1}}^T, ..., \mathbf{b}_{:9}^T$, and the second one storing those of ${\mathbf{b}_{:10}}^T, ..., \mathbf{b}_{:18}^T$. This provides a good trade-off between space and time: while it requires additional computations for row summations, it reduces the amount of memory used for the tables, and also the time to construct them, which also increases exponentially with R.

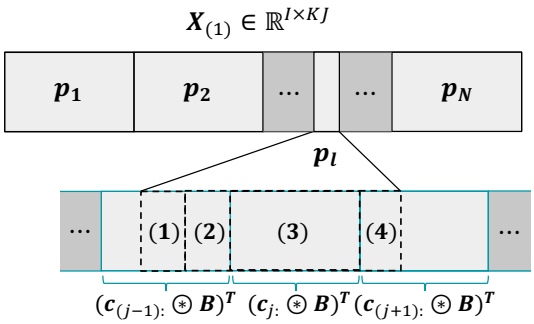


Fig. 5: An overview of partitioning. There are a total of N partitions p_1, p_2, ..., p_N, among which the l-th partition p_l is shown in detail. A partition is divided into "blocks" (rectangles in dashed lines) by the boundaries of underlying pointwise vector-matrix products (blue rectangles). Numbers in p_l refer to the kinds of blocks a partition can be split into.

D. Careful Partitioning of the Workload

The third challenge is how to partition the workload effectively. A partition is a unit of workload distributed across machines. Partitioning is important since it determines the level of parallelism and the amount of shuffled data. Our goal is to fully utilize the available computing resources, while minimizing the amount of network traffic.

First, as described in Section III-B, DBTF partitions the unfolded tensor vertically: a single partition covers a range of consecutive columns. The main reason for choosing vertical partitioning instead of horizontal one is because with vertical partitioning, each partition can perform Boolean summations of the rows assigned to it and compute their errors independently, with no need of communications between partitions. On the other hand, with horizontal partitioning, each partition needs to communicate with others to be able to compute the Boolean row summations. Furthermore, horizontal partitioning splits the range of rank R, which is usually smaller than the dimensionalities of an input tensor; small number of partitions lowers the level of parallelism.

Since the workloads are vertically partitioned, each partition computes an error only for the part of the row distributed to it. Therefore, errors from all partitions should be considered together to make the decision of whether to update an entry to 0 or 1. DBTF collects from all partitions the errors for the entries in the column being updated, and sets each one to the value with the smallest error.

Second, DBTF partitions the unfolded tensor in a cache-friendly manner. By "cache-friendly", we mean structuring the partitions in such a way that facilitates reuse of cached row summation results as discussed in Section III-C. This is crucial since cache utilization affects the performance of DBTF significantly. The unit of caching in DBTF is $\mathbf{B}^T$ as shown in Figure 4. However, the size of each partition is not always the same as or a multiple of that of $\mathbf{B}^T$. Depending on the number of partitions and the sizes of $\mathbf{C}$ and $\mathbf{B}$, a partition

may cross multiple pointwise vector-matrix (PVM) products (i.e., $(\mathbf{c}_{j:} \circledast \mathbf{B})^T$), or may be a part of one PVM product.

Figure 5 presents an overview of our idea for cache-friendly partitioning in DBTF. There are a total of N partitions p_1, p_2, ..., p_N, among which the l-th partition p_l is shown in detail. A partition is divided into "blocks" (rectangles in dashed lines) by the boundaries of underlying PVM products (blue rectangles). Numbers in p_l refer to the kinds of blocks a partition can be split into. Since the unit of caching is $\mathbf{B}^T$, with this orginization, each block of a partition can efficiently fetch its row summation results from the cache table.

Lemma 3. *A partition can have at most three types of blocks.*

Proof. There are four different types of blocks—(1), (2), (3), and (4)—as shown in Figure 5. (a) If the size of a partition is smaller than or equal to that of a single PVM product, it can consist of up to two blocks. When the partition does not cross the boundary of PVM products, it consists of a single block, which corresponds to one of the four types (1), (2), (3), and (4). On the other hand, when the partition crosses the boundary between PVM products, it consists of two blocks of types (2) and (4). (b) If the area covered by a partition is larger than that by a single PVM product, multiple blocks comprise the partition: possible combinations of blocks are (2)+(3)*+(4), (3)$^+$+(4)$^?$, and (2)$^?$+(3)$^+$ where the star (*) superscript denotes that the preceding type is repeated zero or more times, the plus (+) superscript denotes that the preceding type is repeated one or more times, and the question mark (?) superscript denotes that the preceding type is repeated zero or one time. Thus, a partition can have at most three types of blocks. □

An issue that should be considered is that blocks of types (1), (2), and (4) are smaller than a single PVM product. If a partition has such blocks, we compute additional cache tables for the smaller blocks from the full-size one so that these blocks can also exploit caching. By Lemma 3, at most two smaller tables need to be computed for each partition, and each one can be built efficiently as constructing it requires only a single pass over the full-size cache. Partitioning is a one-off task in DBTF; DBTF constructs these partitions in the beginning, and caches the entire partitions for efficiency.

E. Putting things together

We present DBTF in Algorithm 2. DBTF first partitions the unfolded input tensors (lines 1-3 in Algorithm 2): each unfolded tensor is vertically partitioned and then cached (Algorithm 3). Next, DBTF initializes L set of factor matrices randomly (line 6 in Algorithm 2). Instead of initializing a single set of factor matrices, DBTF initializes multiple sets as better initial factor matrices often lead to more accurate factorization. DBTF updates all of them in the first iteration, and runs the following iterations with the factor matrices that obtained the smallest error (lines 7-8 in Algorithm 2). In each iteration, factor matrices are updated one at a time, while the other two are fixed (lines 15-17 in Algorithm 2).

The procedure for updating a factor matrix is shown in Algorithm 4; its core operations—computing a Boolean row summation and its error—are performed in a fully distributed manner (marked by "**D**", lines 7-9). DBTF caches all combinations of Boolean row summations (Algorithm 5) at the beginning of `UpdateFactor` algorithm to avoid repeatedly computing them. Then DBTF collects errors computed across machines, and updates the current column DBTF is visiting (lines 10-12 in Algorithm 4). Boolean factors are repeatedly updated until convergence, that is, until the reconstruction error does not decrease significantly, or a maximum number of iterations has been reached.

Two types of data are sent to each machine: partitions of unfolded tensors are distributed across machines once in the beginning, and factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are broadcast to each machine at each iteration; machines send intermediate errors back to the driver node for each column update.

Algorithm 2: DBTF algorithm

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, rank R, a maximum number of iterations T, a number of sets of initial factor matrices L, and a number of partitions N.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$.

1 ${}_p\mathbf{X}_{(1)} \leftarrow$ `Partition`$(\mathbf{X}_{(1)}, N)$
2 ${}_p\mathbf{X}_{(2)} \leftarrow$ `Partition`$(\mathbf{X}_{(2)}, N)$
3 ${}_p\mathbf{X}_{(3)} \leftarrow$ `Partition`$(\mathbf{X}_{(3)}, N)$
4 **for** $t \leftarrow 1, \ldots, T$ **do**
5 **if** $t = 1$ **then**
6 initialize L sets of factor matrices randomly
7 apply `UpdateFactors` to each set, and find the set s_{min} with the smallest error
8 $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \leftarrow s_{min}$
9 **else**
10 $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \leftarrow$ `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C})$
11 **if** *converged* **then**
12 break out of **for** loop
13 **return** $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
14 **Function** `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C})$
  /* update A to minimize $\left|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^T\right|$ */
15 $\mathbf{A} \leftarrow$ `UpdateFactor`$({}_p\mathbf{X}_{(1)}, \mathbf{A}, \mathbf{C}, \mathbf{B})$
  /* update B to minimize $\left|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^T\right|$ */
16 $\mathbf{B} \leftarrow$ `UpdateFactor`$({}_p\mathbf{X}_{(2)}, \mathbf{B}, \mathbf{C}, \mathbf{A})$
  /* update C to minimize $\left|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^T\right|$ */
17 $\mathbf{C} \leftarrow$ `UpdateFactor`$({}_p\mathbf{X}_{(3)}, \mathbf{C}, \mathbf{B}, \mathbf{A})$
18 **return** $\mathbf{A}, \mathbf{B}, \mathbf{C}$

Algorithm 3: `Partition`

Input: an unfolded binary tensor $\mathbf{X} \in \mathbb{B}^{P \times Q}$, and a number of partitions N.
Output: A partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times Q}$.

1 **Distributed (D):** split $\mathbf{X}$ into non-overlapping partitions $p_1, p_2, \ldots, p_N$ such that $[p_1\ p_2\ \ldots\ p_N] \in \mathbb{B}^{P \times Q}$, and $\forall i \in \{1, ..., N\}, p_i \in \mathbb{B}^{P \times H}$ where $\lfloor \frac{Q}{N} \rfloor \leq H \leq \lceil \frac{Q}{N} \rceil$
2 ${}_p\mathbf{X} \leftarrow [p_1\ p_2\ \ldots\ p_N]$
3 **foreach** $p' \in {}_p\mathbf{X}$ **do**
4 **D:** further split p' into a set of blocks divided by the boundaries of underlying pointwise vector-matrix products as depicted in Figure 5 (see Section III-D)
5 cache ${}_p\mathbf{X}$ across machines
6 **return** ${}_p\mathbf{X}$

Algorithm 4: `UpdateFactor`

Input: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times QS}$, factor matrices $\mathbf{A} \in \mathbb{B}^{P \times R}$ (factor matrix to update), $\mathbf{M}_f \in \mathbb{B}^{Q \times R}$ (first matrix for Khatri-Rao product), and $\mathbf{M}_s \in \mathbb{B}^{S \times R}$ (second matrix for Khatri-Rao product), and a threshold value V to limit the size of a single cache table.

Output: an updated factor matrix $\mathbf{A}$.

1 `CacheRowSummations(`${}_p\mathbf{X}$, $\mathbf{M}_s$, V`)`
`/* iterate over columns and rows of` $\mathbf{A}$ `*/`
2 **for** *column iter* $c \leftarrow 1 \dots R$ **do**
3 **for** *row iter* $r \leftarrow 1 \dots P$ **do**
4 **for** $a_{rc} \leftarrow 0, 1$ **do**
5 **foreach** *partition* $p' \in {}_p\mathbf{X}$ **do**
6 **foreach** *block* $b \in p'$ **do**
7 **Distributed (D):** compute the cache key $k \leftarrow \mathbf{a}_{r:} \wedge [\mathbf{M}_f]_{i:}$ where i is the row index of $\mathbf{M}_f$ such that block b is contained in $([\mathbf{M}_f]_{i:} \circledast \mathbf{M}_s)^T$
8 **D:** using k, fetch the cached Boolean summation of the rows of block b selected by $\mathbf{a}_{r:}$
9 **D:** compute the error between the fetched row and the corresponding part of ${}_p\mathbf{x}_{r:}$
10 collect errors for the entries of column $\mathbf{a}_{:c}$ from all blocks (for both cases of when each entry is set to 0 and 1)
11 **for** *row iter* $r \leftarrow 1 \dots P$ **do** `/* update` $\mathbf{a}_{:c}$ `*/`
12 update a_{rc} to the value that yields a smaller error (i.e., $\left|\mathbf{X}_{r:} - \mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^T\right|$)
13 **return** $\mathbf{A}$

Algorithm 5: `CacheRowSummations`

Input: a partitioned unfolded tensor ${}_p\mathbf{X} \in \mathbb{B}^{P \times QS}$, a matrix for caching $\mathbf{M}_c \in \mathbb{B}^{S \times R}$, and a threshold value V to limit the size of a single cache table.

1 **foreach** *partition* $p' \in {}_p\mathbf{X}$ **do**
2 **Distributed (D):** $m \leftarrow$ all combinations of row summations of $\mathbf{M}_c$ (if $S > V$, divide the rows of $\mathbf{M}_c$ evenly into smaller groups of rows, and cache row summations from each one separately)
3 **foreach** *block* $b \in p'$ **do**
4 **D:** if block b is of the type (1), (2), or (4) as shown in Figure 5, vertically slice m such that the sliced one corresponds to block b
5 **D:** cache (the sliced) m for partition p' if not cached

F. Implementation

In this section, we discuss practical issues pertaining to the implementation of DBTF on Spark. Tensors are loaded as RDDs (*Resilient Distributed Datasets*) [7], and unfolded using RDD's *map* function. We apply *map* and *combineByKey* operations to unfolded tensors for partitioning: *map* transforms an unfolded tensor into a pair RDD whose key is a partition ID; *combineByKey* groups non-zeros by partition ID and organizes them into blocks. Partitioned unfolded tensor RDDs are then *persist*ed in memory. We create a pair RDD containing combinations of row summations, which is keyed by partition ID and *join*ed with the partitioned unfolded tensor RDD. This *join*ed RDD is processed in a distributed manner using *mapPartitions* operation. In obtaining the key to the table for row summations, we use bitwise AND operation for efficiency. At the end of column-wise iteration, a driver node *collect*s errors computed from each partition to update the column.

G. Analysis

We analyze the proposed method in terms of time complexity, memory requirement, and the amount of shuffled data. We use the following symbols in the analysis: R (rank), M (number of machines), T (number of maximum iterations), L (number of sets of initial factor matrices), N (number of partitions), and V (maximum number of rows to be cached). For the sake of simplicity, we assume an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$.

Lemma 4. *The time complexity of* DBTF *is* $O(|\mathcal{X}| + (L + T)\big(N\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil} I + IR\big\lceil \lceil \frac{R}{V} \rceil (\min(V, R) \max(I, N) + I^2) + N \big\rceil\big)$.

Proof. Algorithm 2 is composed of three operations: (1) partitioning (lines 1-3), (2) initialization (line 6), and (3) updating factor matrices (lines 7 and 10). (1) After unfolding an input tensor $\mathcal{X}$ into $\mathbf{X}$, DBTF splits $\mathbf{X}$ into N partitions, and further divides each partition into a set of blocks (Algorithm 3). Unfolding takes $O(|\mathcal{X}|)$ time as each entry can be mapped in constant time (Equation 1), and partitioning takes $O(|\mathbf{X}|)$ time since determining which partition and block an entry of $\mathbf{X}$ belongs to is also a constant-time operation. It takes $O(|\mathcal{X}|)$ time in total. (2) Randomly initializing L sets of factor matrices takes $O(LIR)$ time. (3) The update of a factor matrix (Algorithm 4) consists of four steps as follows.

i. Caching row summations (line 1). By Lemma 2, the number of cache tables is $\lceil R/V \rceil$, and the maximum size of a single cache table is $2^{\lceil R/\lceil R/V \rceil \rceil}$. Each row summation can be obtained in $O(I)$ time via incremental computations that use prior row summation results. Hence, caching row summations for N partitions takes $O(N\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V \rceil \rceil} I)$.
ii. Fetching a cached row summation (lines 7-8). The number of constructing row summations and computing errors to update a factor matrix is $2IR$. An entire row summation is constructed by fetching row summations from the cache tables $O(\max(I, N))$ times across N partitions. If $R \leq V$, a row summation can be constructed by a single access to the cache. If $R > V$, multiple accesses are required to fetch row summations from $\lceil \frac{R}{V} \rceil$ tables. Also, constructing a cache key requires $O(\min(V, R))$ time. Thus, fetching a cached row summation takes $O(\lceil \frac{R}{V} \rceil \min(V, R) \max(I, N))$ time. When $R > V$, there is an additional cost to sum up $\lceil \frac{R}{V} \rceil$ row summations, which is $O((\lceil \frac{R}{V} \rceil - 1)I^2)$. In total, it takes $O(IR\big[\lceil \frac{R}{V} \rceil \min(V, R) \max(I, N) + (\lceil \frac{R}{V} \rceil - 1)I^2\big])$.
iii. Computing the error for the fetched row summation (line 9). It takes $O(I^2)$ time to calculate an error of one row summation with regard to the corresponding row of the unfolded tensor. For each column entry, DBTF constructs row summations ($\mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^T$ in Algorithm 4) twice (for $a_{rc} = 0$ and 1). Therefore, given a rank R, this step takes $O(I^3 R)$ time. Note that the time complexities for

steps ii and iii are a loose upper bound since in practice the computations for the I^2 terms take time proportional to the number of non-zeros in the involved matrices.

iv. Updating a factor matrix (lines 10-12). Updating an entry in a factor matrix requires summing up errors for each value that are collected from all partitions; this takes $O(N)$ time. Updating all entries takes $O(NIR)$ time.

Thus, DBTF takes $O(|\mathcal{X}| + (L+T)\big(N\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I + IR\big\lceil\frac{R}{V}\big\rceil(\min(V,R)\max(I,N)+I^2)+N\big\rceil\big)$ time. □

Lemma 5. *The memory requirement of* DBTF *is* $O(|\mathcal{X}| + NI\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} + MRI)$.

Proof. For the decomposition of an input tensor $\mathcal{X} \in \mathbb{B}^{I\times I\times I}$, DBTF stores the following four types of data in memory at each iteration: (1) partitioned unfolded tensors ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$, (2) row summation results, (3) factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, and (4) errors for the entries of a column being updated. (1) While partitioning of an unfolded tensor by DBTF structures it differently from the original one, the total number of elements does not change after partitioning. Thus, ${}_p\mathbf{X}_{(1)}, {}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ require $O(|\mathcal{X}|)$ memory. (2) By Lemma 2, the total number of cached row summations is $O(\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$. By Lemma 3, each partition has at most three types of blocks. Since an entry in the cache table uses $O(I)$ space, the total amount of memory used for row summation results is $O(NI\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$. Note that since Boolean factor matrices are normally sparse, many cached row summations are not normally dense. Therefore, the actual amount of memory used is usually smaller than the stated upper bound. (3) Since $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ are broadcast to each machine, they require $O(MRI)$ memory in total. (4) Each partition stores the errors for the entries of the column being updated, which takes $O(NI)$ memory. □

Lemma 6. *The amount of shuffled data for partitioning an input tensor* $\mathcal{X}$ *is* $O(|\mathcal{X}|)$.

Proof. DBTF unfolds an input tensor $\mathcal{X}$ into three different modes, $\mathbf{X}_{(1)}$, $\mathbf{X}_{(2)}$, and $\mathbf{X}_{(3)}$, and then partitions each one: unfolded tensors are shuffled across machines so that each machine has a specific range of consecutive columns of unfolded tensors. In the process, the entire data can be shuffled, depending on the initial distribution of the data. Thus, the amount of data shuffled for partitioning $\mathcal{X}$ is $O(|\mathcal{X}|)$. □

Lemma 7. *The amount of shuffled data after the partitioning of an input tensor* $\mathcal{X}$ *is* $O(TRI(M+N))$.

Proof. Once the three unfolded input tensors $\mathbf{X}_{(1)}$, $\mathbf{X}_{(2)}$, and $\mathbf{X}_{(3)}$ are partitioned, they are cached across machines, and are not shuffled. In each iteration, DBTF broadcasts three factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ to each machine, which takes $O(MRI)$ space in sum. With only these three matrices, each machine generates the part of row summation it needs to process. Also, in updating a factor matrix of size I-by-R, DBTF collects from all partitions the errors for both cases of when each entry of the factor matrix is set to 0 and 1. This process involves transmitting $2IR$ errors from each partition to the driver node, which takes $O(NRI)$ space in total. Accordingly, the total amount of data shuffled for T iterations after partitioning $\mathcal{X}$ is $O(TRI(M+N))$. □

TABLE III: Summary of real-world and synthetic tensors used for experiments. B: billion, M: million, K: thousand.

Name	I	J	K	Non-zeros
Facebook	64K	64K	870	1.5M
DBLP	418K	3.5K	50	1.3M
CAIDA-DDoS-S	9K	9K	4K	22M
CAIDA-DDoS-L	9K	9K	393K	331M
NELL-S	15K	15K	29K	77M
NELL-L	112K	112K	213K	18M
Synthetic-scalability	2^6∼2^{13}	2^6∼2^{13}	2^6∼2^{13}	26K∼5.5B
Synthetic-error	100	100	100	7K∼240K

IV. Experiments

In this section, we experimentally evaluate our proposed method DBTF. We aim to answer the following questions.

- **Q1 Data Scalability (Section IV-B).** How well do DBTF and other methods scale up with respect to the following aspects of an input tensor: number of non-zeros, dimensionality, density, and rank?
- **Q2 Machine Scalability (Section IV-C).** How well does DBTF scale up with respect to the number of machines?
- **Q3 Reconstruction error (Section IV-D).** How accurately do DBTF and other methods factorize the given tensor?

We introduce the datasets and experimental environment in Section IV-A. After that, we answer the above questions in Sections IV-B to IV-D.

A. Experimental Settings

1) Datasets: We use both real-world and synthetic tensors to evaluate the proposed method. The tensors used in experiments are listed in Table III. For real-world tensors, we use Facebook, DBLP, CAIDA-DDoS-S, CAIDA-DDoS-L, NELL-S, and NELL-L. Facebook[1] is temporal relationship data between users. DBLP[2] is a record of DBLP publications. CAIDA-DDoS[3] datasets are traces of network attack traffic. NELL datasets are knowledge base tensors; S (small) and L (large) suffixes indicate the relative size of the dataset.

We prepare two different sets of synthetic tensors, one for scalability tests and another for reconstruction error tests. For scalability tests, we generate random tensors, varying the following aspects: (1) dimensionality and (2) density. We vary one aspect while fixing others to see how scalable DBTF and other methods are with respect to a particular aspect. For reconstruction error tests, we generate three random factor matrices, construct a noise-free tensor from them, and then add noise to this tensor, while varying the following aspects: (1) factor matrix density, (2) rank, (3) additive noise level, and (4) destructive noise level. When we vary one aspect, others are fixed. The amount of noise is determined by the number of

[1] http://socialnetworks.mpi-sws.org/data-wosn2009.html
[2] http://www.informatik.uni-trier.de/~ley/db/
[3] http://www.caida.org/data/passive/ddos-20070804_dataset.xml

1's in the noise-free tensor. For example, 10% additive noise indicates that we add 10% more 1's to the noise-free tensor, and 5% destructive noise means that we delete 5% of the 1's from the noise-free tensor.

2) Environment: DBTF is implemented on Spark, and compared with two previous algorithms for Boolean CP decomposition: Walk'n'Merge [2] and BCP_ALS [3]. We run experiments on a cluster with 17 machines, each of which is equipped with an Intel Xeon E3-1240v5 CPU (quad-core with hyper-threading at 3.50GHz) and 32GB RAM. The cluster runs Spark v2.0.0, and consists of a driver node and 16 worker nodes. In the experiments for DBTF, we use 16 executors, and each executor uses 8 cores. The amount of memory for the driver and each executor process is set to 16GB and 25GB, respectively. The default values for DBTF parameters L, V, and T (see Algorithms 2-5) are set to 1, 15, and 10, respectively. We run Walk'n'Merge and BCP_ALS on one machine in the cluster. For Walk'n'Merge, we use the original implementation[4] provided by the authors, and run it with the same parameter settings as in [2] to get similar results: the merging threshold t is set to $1 - (n_d + 0.05)$ where n_d is the destructive noise level of an input tensor; the minimum size of blocks is 4-by-4-by-4; the length of random walks is 5; the other parameters are set to default values. We implement BCP_ALS using the open-source code of ASSO[5][8]. For ASSO, the threshold value for discretization is set to 0.7; default values are used for other parameters.

B. Data Scalability

We evaluate the data scalability of DBTF and other methods using both synthetic random and real-world tensors.

1) Synthetic Data: With synthetic tensors, we measure the data scalability with regard to three different criteria. We allow experiments to run for up to 6 hours, and mark those running longer than that as O.O.T. (Out-Of-Time).

Dimensionality. We increase the dimensionality $I{=}J{=}K$ of each mode from 2^6 to 2^{13}, while setting the tensor density to 0.01 and the rank to 10. As shown in Figure 1(a), DBTF successfully decomposes tensors of size $I{=}J{=}K{=}2^{13}$, while Walk'n'Merge and BCP_ALS run out of time when $I{=}J{=}K \geq 2^9$ and $\geq 2^{10}$, respectively. Notice that the running time of Walk'n'Merge and BCP_ALS increases rapidly with the dimensionality: DBTF decomposes the largest tensors Walk'n'Merge and BCP_ALS can process 382 times and 68 times faster than each method. Only for the smallest tensor of 2^6 scale, DBTF is slower than other methods, which is because the overhead of running a distributed algorithm on Spark (e.g., code and data distribution, network I/O latency, etc) dominates the running time.

Density. We increase the tensor density from 0.01 to 0.3, while fixing $I{=}J{=}K$ to 2^8 and the rank to 10. As shown in Figure 1(b), DBTF decomposes tensors of all densities, and exhibits near constant performance regardless of the density.

[4]http://people.mpi-inf.mpg.de/~pmiettin/src/walknmerge.zip
[5]http://people.mpi-inf.mpg.de/~pmiettin/src/DBP-progs/

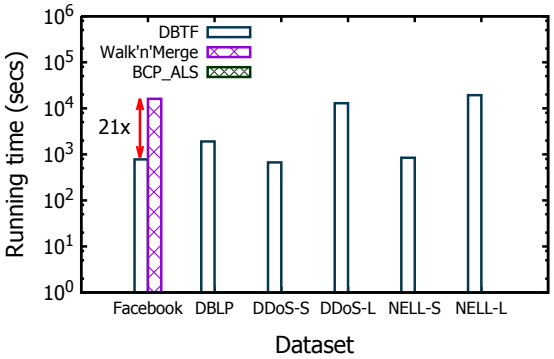


Fig. 6: The scalability of DBTF and other methods on the real-world datasets. Notice that only DBTF scales up to all datasets, while Walk'n'Merge processes only Facebook, and BCP_ALS fails to process all datasets. DBTF runs 21× faster than Walk'n'Merge on Facebook. An empty bar denotes that the corresponding method runs out of time (> 12 hours) or memory while decomposing the dataset.

BCP_ALS also scales up to 0.3 density. On the other hand, Walk'n'Merge runs out of time when the density increases over 0.1. In terms of running time, DBTF runs 716 times faster than Walk'n'Merge, and 13 times faster than BCP_ALS. This relatively small difference between the running times of DBTF and BCP_ALS is due to the small dimensionality of the tensor; for tensors with larger dimensionalities, the performance gap between the two grows wider as we see in Figure 1(a).

Rank. We increase the rank of a tensor from 10 to 60, while fixing $I{=}J{=}K$ to 2^8 and the tensor density to 0.01. As shown in Figure 1(c), while all methods scale up to rank 60, DBTF is the fastest among them: DBTF is 21 times faster than BCP_ALS, and 43 times faster than Walk'n'Merge when the rank is 60. Note that V is set to 15 in all experiments. Since Walk'n'Merge returns more than 60 dense blocks (rank-1 tensors) from the input tensor, the running time of Walk'n'Merge is the same across all ranks.

2) Real-world Data: We measure the running time of each method on the real-world datasets. For real-world tensors, we set the maximum running time to 12 hours. As Figure 6 shows, DBTF is the only method that scales up for all datasets. Walk'n'Merge decomposes only Facebook, and runs out of time for all other datasets; BCP_ALS fails to handle real-world tensors as it causes out-of-memory (O.O.M.) errors for all datasets, except for DBLP for which BCP_ALS runs out of time. Also, DBTF runs 21 times faster than Walk'n'Merge on Facebook.

C. Machine Scalability

We measure the machine scalability by increasing the number of machines from 4 to 16, and report T_4/T_M where T_M is the running time using M machines. We use the synthetic tensor of size $I{=}J{=}K{=}2^{12}$ and of density 0.01, and set the rank to 10. As Figure 7 shows, DBTF shows near linear scalability, achieving 2.2× speed-up when the number of machines is increased from 4 to 16.

D. Reconstruction Error

We evaluate the accuracy of DBTF in terms of reconstruction error, which is defined as $|\mathcal{X} - \mathcal{X}'|$ where $\mathcal{X}$ is an

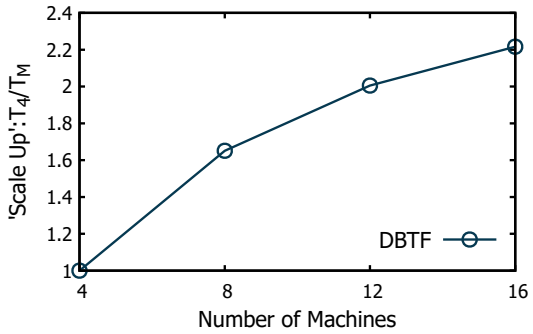


Fig. 7: The scalability of DBTF with respect to the number of machines. T_M means the running time using M machines. Notice that the running time scales up near linearly.

input tensor and $\mathcal{X}'$ is a reconstructed tensor. In measuring reconstruction errors, we vary one of the four different data aspects—factor matrix density (0.1), rank (10), additive noise level (0.1), and destructive noise level (0.1)—while fixing the others to the default values. The values in the parentheses are the default settings for each aspect. Tensors of size $I{=}J{=}K{=}100$ are used in experiments. The DBTF parameter L is set to 20. We run each configuration three times, and report the average of the results to reduce the dependency on randomness of DBTF and Walk'n'Merge. We compare DBTF with Walk'n'Merge as they take different approaches for Boolean CP decomposition, and exclude BCP_ALS as DBTF and BCP_ALS are based on the same Boolean CP decomposition framework (Algorithm 1). For Walk'n'Merge, we compute the reconstruction error from the blocks obtained before the merging phase [2], since the merging procedure significantly increased the reconstruction error when applied to our synthetic tensors. Figure 8(d) shows the difference between the version of Walk'n'Merge with the merging procedure (Walk'n'Merge*) and the one without it (Walk'n'Merge).

Factor Matrix Density. We increase the density of factor matrices from 0.1 to 0.3. As shown in Figure 8(a), the reconstruction error of DBTF is smaller than that of Walk'n'Merge for all densities. In particular, as the density increases, DBTF obtains more accurate results compared to Walk'n'Merge.

Rank. We increase the rank of a tensor from 10 to 60. As shown in Figure 8(b), the reconstruction errors of both methods increase in proportion to the rank. This is an expected result since, given a fixed density, the increase in the rank of factor matrices leads to increased number of 1's in the input tensor. Notice that the reconstruction error of DBTF is smaller than that of Walk'n'Merge for all ranks.

Additive Noise Level. We increase additive noise level from 0.1 to 0.4. As Figure 8(c) shows, the reconstruction errors of both methods increase in proportion to the additive noise level. While the gap between the two methods narrows down as the noise level increases, the reconstruction error of DBTF is smaller than that of Walk'n'Merge for all additive noise levels.

Destructive Noise Level. We increase destructive noise level from 0.1 to 0.4. As Figure 8(d) shows, the reconstruction errors of DBTF and Walk'n'Merge decrease in general as the destructive noise level increases, except for the interval from 0.1 to 0.2 where that of DBTF increases. Destructive noise makes the factorization harder by sparsifying tensors and introducing noises at the same time. Notice that DBTF produces more accurate results than Walk'n'Merge, except at the destructive noise level 0.4, which makes tensors highly sparse.

V. Related Works

In this section, we review related works on Boolean and normal tensor decompositions, and distributed computing frameworks.

A. Boolean tensor decomposition.

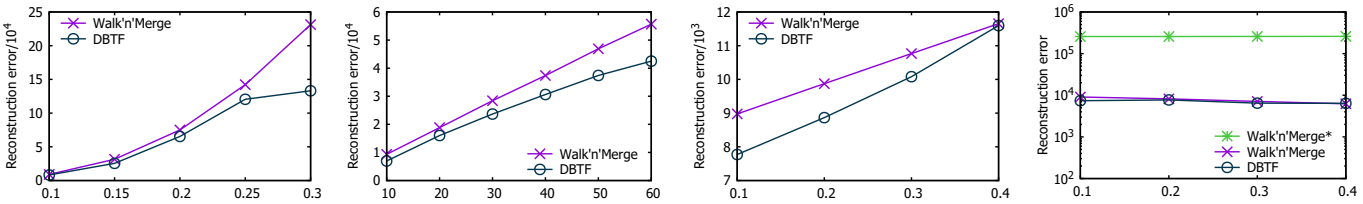
Leenen et al. [6] proposed the first Boolean CP decomposition algorithm. Miettinen [3] presented Boolean CP and Tucker decomposition methods along with a theoretical study of Boolean tensor rank and decomposition. In [5], Belohlávek et al. presented a greedy algorithm for Boolean CP decomposition of three-way binary data. Erdös et al. [2] proposed a scalable algorithm for Boolean CP and Tucker decompositions, which performs random walk for finding dense blocks (rank-1 tensors) and applies the MDL principle to select the best rank automatically. In [9], Erdös et al. applied the Boolean Tucker decomposition method proposed in [2] to discover synonyms and find facts from the subject-predicate-object triples. Finding closed itemsets in N-way binary tensor [10], [11] is a restricted form of Boolean CP decomposition, in which an error of representing 0's as 1's is not allowed. Metzler et al. [4] presented an algorithm for Boolean tensor clustering, which is another form of restricted Boolean CP decomposition where one of the factor matrices has exactly one non-zero per row.

B. Normal tensor decomposition.

Many algorithms have been developed for normal tensor decomposition. In this subsection, we focus on scalable approaches developed recently. GigaTensor [12] is the first work for large-scale CP decomposition running on MapReduce. HaTen2 [13], [14] improves upon GigaTensor and presents a general, unified framework for Tucker and CP decompositions. In [15], Jeon et al. proposed SCouT for scalable coupled matrix-tensor factorization. Recently, tensor decomposition methods proposed in [12], [16], [13], [14], [15] have been integrated into a multi-purpose tensor mining library, BIGtensor [17]. Beutel et al. [18] proposed FlexiFaCT, a scalable MapReduce algorithm to decompose matrix, tensor, and coupled matrix-tensor using stochastic gradient descent. CDTF [19], [20] provides a scalable tensor factorization method focusing on non-zero elements of tensors.

C. Distributed computing frameworks.

MapReduce [21] is a distributed programming model for processing large datasets in a massively parallel manner. The advantages of MapReduce include massive scalability, fault tolerance, and automatic data distribution and replication. Hadoop [22] is an open source implementation of MapReduce. Due to the advantages of MapReduce, many data mining tasks [12], [23], [24], [25] have used Hadoop. However, due to



(a) Factor Matrix Density (b) Rank (c) Additive Noise Level (d) Destructive Noise Level

Fig. 8: The reconstruction error of DBTF and other methods with respect to factor matrix density, rank, additive noise level, and destructive noise level. Walk'n'Merge* in (d) refers to the version of Walk'n'Merge which executes the merging phase. Notice that the reconstruction errors of DBTF are smaller than those of Walk'n'Merge for all aspects except for the tensor with the largest destructive noise.

intensive disk I/O, Hadoop is inefficient at executing iterative algorithms [26]. Spark [7] is a distributed data processing framework that provides capabilities for in-memory computation and data storage. These capabilities enable Spark to perform iterative computations efficiently, which are common across many machine learning and data mining algorithms, and support interactive data analytics. Spark also supports various operations other than *map* and *reduce*, such as *join*, *filter*, and *groupBy*. Thanks to these advantages, Spark has been used in several domains recently [27], [28], [29], [30].

VI. Conclusion

In this paper, we propose DBTF, a distributed algorithm for Boolean tensor factorization. By caching computation results, exploiting the characteristics of Boolean operations, and with careful partitioning, DBTF successfully tackles the high computational costs and minimizes the intermediate data. Experimental results show that DBTF decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 68–382$\times$ less time, and exhibits near-linear scalability in terms of tensor dimensionality, density, rank, and machines.

Future works include extending the method for other Boolean tensor decomposition methods including Boolean Tucker.

Acknowledgment

This work was supported by the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (NRF-2016M3C4A7952587, PF Class Heterogeneous High Performance Computer Development). U Kang is the corresponding author.

References

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.
[2] D. Erdös and P. Miettinen, "Walk 'n' merge: A scalable algorithm for boolean tensor factorization," in *ICDM*, 2013, pp. 1037–1042.
[3] P. Miettinen, "Boolean tensor factorizations," in *ICDM*, 2011, pp. 447–456.
[4] S. Metzler and P. Miettinen, "Clustering boolean tensors," *DMKD*, vol. 29, no. 5, pp. 1343–1373, 2015.
[5] R. Belohlávek, C. V. Glodeanu, and V. Vychodil, "Optimal factorization of three-way binary data using triadic concepts," *Order*, vol. 30, no. 2, pp. 437–454, 2013.
[6] I. Leenen, I. Van Mechelen, P. De Boeck, and S. Rosenberg, "Indclas: A three-way hierarchical classes model," *Psychometrika*, vol. 64, no. 1, pp. 9–24, 1999.
[7] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI*, 2012, pp. 15–28.
[8] P. Miettinen, T. Mielikäinen, A. Gionis, G. Das, and H. Mannila, "The discrete basis problem," *TKDE*, vol. 20, no. 10, pp. 1348–1362, 2008.
[9] D. Erdös and P. Miettinen, "Discovering facts with boolean tensor tucker decomposition," in *CIKM*, 2013, pp. 1569–1572.
[10] L. Cerf, J. Besson, C. Robardet, and J. Boulicaut, "Closed patterns meet *n*-ary relations," *TKDD*, vol. 3, no. 1, 2009.
[11] L. Ji, K. Tan, and A. K. H. Tung, "Mining frequent closed cubes in 3d datasets," in *VLDB*, 2006, pp. 811–822.
[12] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries," in *KDD*, 2012, pp. 316–324.
[13] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," in *ICDE*, 2015, pp. 1047–1058.
[14] I. Jeon, E. E. Papalexakis, C. Faloutsos, L. Sael, and U. Kang, "Mining billion-scale tensors: algorithms and discoveries," *VLDB J.*, vol. 25, no. 4, pp. 519–544, 2016.
[15] B. Jeon, I. Jeon, L. Sael, and U. Kang, "Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries," in *ICDE*, 2016, pp. 811–822.
[16] L. Sael, I. Jeon, and U. Kang, "Scalable tensor mining," *Big Data Research*, vol. 2, no. 2, pp. 82 – 86, 2015, visions on Big Data.
[17] N. Park, B. Jeon, J. Lee, and U. Kang, "Bigtensor: Mining billion-scale tensor made easy," in *CIKM*, 2016, pp. 2457–2460.
[18] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing, "Flexifact: Scalable flexible factorization of coupled tensors on hadoop," in *SDM*, 2014, pp. 109–117.
[19] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *ICDM*, 2014.
[20] K. Shin, L. Sael, and U. Kang, "Fully scalable methods for distributed tensor factorization," *TKDE*, vol. 29, no. 1, pp. 100–113, 2017.
[21] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.
[22] "Apache hadoop," http://hadoop.apache.org/.
[23] H.-M. Park, N. Park, S.-H. Myaeng, and U. Kang, "Partition aware connected component computation in distributed systems," in *ICDM*, 2016.
[24] H.-M. Park, S.-H. Myaeng, and U. Kang, "Pte: Enumerating trillion triangles on distributed systems," in *KDD*, 2016, pp. 1115–1124.
[25] U. Kang, H. Tong, J. Sun, C. Lin, and C. Faloutsos, "GBASE: a scalable and general graph management system," in *KDD*, 2011, pp. 1091–1099.
[26] V. Kalavri and V. Vlassov, "Mapreduce: Limitations, optimizations and open issues," in *TrustCom*, 2013, pp. 1031–1038.
[27] A. Lulli, L. Ricci, E. Carlini, P. Dazzi, and C. Lucchese, "Cracker: Crumbling large graphs into connected components," in *ISCC*, 2015, pp. 574–581.
[28] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, vol. 30, no. 18, pp. 2652–2653, 2014.
[29] R. B. Zadeh, X. Meng, A. Ulanov, B. Yavuz, L. Pu, S. Venkataraman, E. R. Sparks, A. Staple, and M. Zaharia, "Matrix computations and optimization in apache spark," in *KDD*, 2016, pp. 31–38.
[30] H. Kim, J. Park, J. Jang, and S. Yoon, "Deepspark: Spark-based deep learning supporting asynchronous updates and caffe compatibility," *CoRR*, vol. abs/1602.08191, 2016.