Fast and scalable method for distributed Boolean tensor factorization

Namyong Park, Sejoon Oh & U Kang

The VLDB Journal
The International Journal on Very Large Data Bases

ISSN 1066-8888

The VLDB Journal
DOI 10.1007/s00778-019-00538-z

Springer

The VLDB Journal
https://doi.org/10.1007/s00778-019-00538-z

REGULAR PAPER

Fast and scalable method for distributed Boolean tensor factorization

Namyong Park[1] · Sejoon Oh[2] · U Kang[3]

Received: 7 February 2018 / Revised: 3 October 2018 / Accepted: 21 December 2018

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 method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations running on the Apache Spark framework. By distributed data generation with minimal network transfer, 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-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180 × less time, and DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129 × less time. Furthermore, both DBTF-CP and DBTF-TK exhibit near-linear scalability in terms of tensor dimensionality, density, rank, and machines.

Keywords Tensor · Tensor factorization · Boolean CP factorization · Boolean Tucker factorization · Distributed algorithm

1 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 0's and 1's. 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 nonnegativity 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 latent concept discovery, clustering, recommendation, link prediction, and synonym finding. For example, low-rank BTF yields factor matrices whose columns correspond to latent concepts underlying the data, and applying a Boolean Tucker factorization to subject–predicate–object triples can find synonyms and uncover facts from the data [4].

✉ U Kang
ukang@snu.ac.kr

Namyong Park
namyongp@cs.cmu.edu

Sejoon Oh
ohhenrie@cmu.edu

[1] Computer Science Department, Carnegie Mellon University, Pittsburgh, USA

[2] Computational Biology Department, Carnegie Mellon University, Pittsburgh, USA

[3] Department of Computer Science and Engineering, Seoul National University, Seoul, Korea

Published online: 18 March 2019

Table 1 Comparison of our proposed DBTF and existing methods for Boolean (a) CP and (b) Tucker factorizations in terms of whether a method is parallel and distributed. DBTF is the only approach that is parallel and distributed

Method	Parallel	Distributed
(a) Boolean CP Factorization		
Walk'n'Merge [2]	Yes	No
BCP_ALS [3]	No	No
DBTF-CP	Yes	Yes
(b) Boolean Tucker Factorization		
Walk'n'Merge [2]	Yes	No
BTucker_ALS [3]	No	No
DBTF-TK	Yes	Yes

BTF requires that the input tensor, all factor matrices, and a core tensor are binary. Furthermore, BTF uses Boolean sum instead of normal addition, which means $1 + 1 = 1$ in BTF. When the data are inherently 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 nonnegativity constrained counterparts, though at the expense of increased computational complexity [3,5].

While several algorithms have been developed for BTF [2,3,6,7], they are not fast and scalable enough for large-scale tensors that have become widespread. For example, consider DBLP and NELL datasets, which are two different types of real-world tensors consisting of 1.3M to 77M nonzeros. In our experiments on these tensors, all of the state-of-the-art BTF methods get terminated due to out-of-memory errors, or failed at processing them within a reasonable amount of time. Even in the case where existing approaches could be applied, their performance is not enough for many practical applications. In order for BTF to be used for the analysis of large-scale tensors in practical settings, it is of great importance to overcome these limitations. In summary, 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.

In this paper, we propose DBTF (distributed Boolean tensor factorization), a distributed method for Boolean CP and Tucker factorizations running on the Apache Spark framework [8]. DBTF tackles the high computational cost by reducing the operations involved with BTF in an efficient greedy algorithm, 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 1 shows a comparison of DBTF and existing methods in terms of whether a method is parallel and distributed. Note that DBTF is the only approach that is parallel and distributed.

The main contributions of this paper are as follows:

- **Algorithm** We propose DBTF, a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations, which is designed to scale up to large tensors by minimizing intermediate data and the number of operations for BTF, and carefully partitioning the workload.
- **Theory** We provide an analysis of the proposed DBTF-CP and DBTF-TK 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. We show that the proposed Boolean CP factorization method decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180× less time, and the proposed Boolean Tucker factorization method decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129× less time. We also show that DBTF successfully decomposes real-world tensors, such as DBLP and NELL, that cannot be processed with the state-of-the-art BTF methods.

The code of our method and the datasets used in this paper are available at https://www.cs.cmu.edu/~namyongp/dbtf/. The preliminary version of this work is described in [9]. In this paper, we present a distributed method for Boolean Tucker factorization (DBTF-TK) in Sects. 4.1–4.7, in addition to the Boolean CP factorization method (DBTF-CP) previously presented in [9]. We also provide a theoretical analysis, and an experimental evaluation of DBTF-TK in Sects. 4.8 and 5, respectively.

The rest of the paper is organized as follows. We present the preliminaries of CP and Tucker factorizations in normal and Boolean settings in Sect. 2. Then, we discuss related works in Sect. 3 and describe our proposed method for fast and scalable Boolean CP and Tucker factorization in Sect. 4. After presenting experimental results in Sect. 5, we conclude in Sect. 6.

2 Preliminaries

In this section, we provide the definition of Boolean arithmetic and present the notations and operations used for tensor decomposition. Next, we give the definitions of normal CP

and Tucker decompositions, and those of Boolean CP and Tucker decompositions. After that, we introduce approaches for computing Boolean CP and Tucker decompositions. Symbols used in the paper are summarized in Table 2.

2.1 Boolean arithmetic

Given binary data, all operations involved with Boolean tensor factorization operate with Boolean arithmetic in which addition (Boolean OR which is denoted by $\vee$) and multiplication (Boolean AND which is denoted by $\wedge$) between two variables are defined as:

x	y	$x \wedge y$	$x \vee y$
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

2.2 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).

Tensors and matrices Tensor is a multi-dimensional array. The dimension of a tensor is also referred to as *mode*, *order*, or *way*. A matrix $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$ is a tensor of order two. 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 $(i_1, i_2, \ldots, i_N)$th entry of a tensor $\mathcal{X}$ is denoted by $x_{i_1 i_2 \ldots i_N}$. A colon in the subscript indicates taking all elements of that mode. For example, given a matrix $\mathbf{A}$, $\mathbf{a}_{:j}$ denotes the jth column, and $\mathbf{a}_{i:}$ denotes the ith row. The jth column of $\mathbf{A}$, $\mathbf{a}_{:j}$, is also denoted more concisely as $\mathbf{a}_j$. A colon between two numbers in the subscript denotes taking all such elements whose indices in that mode lie between the given numbers. For instance, $\mathbf{A}_{(1:5)j}$ indicates the first five elements of $\mathbf{a}_j$. For a three-way tensor $\mathcal{X}$, $\mathbf{x}_{:jk}$, $\mathbf{x}_{i:k}$, and $\mathbf{x}_{ij:}$ are called column (mode-1), row (mode-2), and tube (mode-3) fibers, respectively. $|\mathcal{X}|$ denotes the number of nonzero elements in a tensor $\mathcal{X}$; $\|\mathcal{X}\|$ denotes the Frobenius norm of a tensor $\mathcal{X}$ and is defined as $\sqrt{\sum_{i,j,k} x_{ijk}^2}$.

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:

Table 2 Table of symbols

Symbol	Definition
$\mathcal{X}$	Tensor (Euler script, bold letter)
$\mathbf{A}$	Matrix (in uppercase, bold letter)
$\mathbf{a}$	Column vector (lowercase, bold letter)
a	Scalar (lowercase, italic letter)
R	Rank (number of components)
$\mathcal{G}$	Core tensor ($\in \mathbb{R}^{R_1 \times R_2 \times R_3}$)
$\mathbf{X}_{(n)}$	Mode-n matricization of a tensor $\mathcal{X}$
$\|\mathcal{X}\|$	Number of nonzeros in the tensor $\mathcal{X}$
$\|\mathcal{X}\|$	Frobenius norm of the tensor $\mathcal{X}$
$\mathbf{A}^\top$	Transpose of matrix $\mathbf{A}$
$\circ$	Outer product
$\otimes$	Kronecker product
$\odot$	Khatri–Rao 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 $\mathcal{X}$
R_1, R_2, R_3	Dimensions of each mode of a core tensor $\mathcal{G}$

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

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:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1J_1}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2J_1}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I_1 1}\mathbf{B} & a_{I_1 2}\mathbf{B} & \cdots & a_{I_1 J_1}\mathbf{B} \end{bmatrix}. \tag{2}$$

The Kronecker product of matrices $\mathbf{A} \in \mathbb{R}^{I_1 \times J_1}$ and $\mathbf{B} \in \mathbb{R}^{I_2 \times J_2}$ can also be expressed by vector-matrix Kronecker products as follows:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} \mathbf{a}_{1:} \otimes \mathbf{B} \\ \mathbf{a}_{2:} \otimes \mathbf{B} \\ \vdots \\ \mathbf{a}_{I_1:} \otimes \mathbf{B} \end{bmatrix}. \tag{3}$$

Khatri–Rao product The Khatri–Rao product (or column-wise Kronecker product) of matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$ produces a matrix of size IJ-by-R and is defined as:

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_{:1} \otimes \mathbf{b}_{:1} \ \mathbf{a}_{:2} \otimes \mathbf{b}_{:2} \ \ldots \ \mathbf{a}_{:R} \otimes \mathbf{b}_{:R}]. \tag{4}$$

Given matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$ and $\mathbf{B} \in \mathbb{R}^{J \times R}$, the Khatri–Rao product $\mathbf{A} \odot \mathbf{B}$ can also be expressed by vector-matrix Khatri–Rao products as follows:

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} \mathbf{a}_{1:} \odot \mathbf{B} \\ \mathbf{a}_{2:} \odot \mathbf{B} \\ \vdots \\ \mathbf{a}_{I_1:} \odot \mathbf{B} \end{bmatrix}. \tag{5}$$

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 is 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:

$$(\mathcal{X} \vee \mathcal{Y})_{ijk} = x_{ijk} \vee y_{ijk}. \tag{6}$$

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:

$$(\mathbf{A} \boxtimes \mathbf{B})_{ij} = \bigvee_{k=1}^{R} a_{ik} b_{kj}. \tag{7}$$

2.3 Tensor rank and tensor decompositions

2.3.1 Normal tensor rank and tensor decompositions

With the above notations, we first give the definitions of normal tensor rank, and normal CP and Tucker decompositions.

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.,

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r. \tag{8}$$

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

$$\left\| \mathcal{X} - \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right\|. \tag{9}$$

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

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}(\mathbf{C} \odot \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B}(\mathbf{C} \odot \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C}(\mathbf{B} \odot \mathbf{A})^\top. \end{aligned} \tag{10}$$

Definition 3 (*Tucker decomposition*) Given a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and the dimensions of a core tensor R_1, R_2, and R_3, find factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R_1}$, $\mathbf{B} \in \mathbb{R}^{J \times R_2}$, $\mathbf{C} \in \mathbb{R}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ such that they minimize

$$\left\| \mathcal{X} - \sum_{r_1=1}^{R_1} \sum_{r_2=1}^{R_2} \sum_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3} \right\|. \tag{11}$$

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

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A}\mathbf{G}_{(1)}(\mathbf{C} \otimes \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B}\mathbf{G}_{(2)}(\mathbf{C} \otimes \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C}\mathbf{G}_{(3)}(\mathbf{B} \otimes \mathbf{A})^\top. \end{aligned} \tag{12}$$

2.3.2 Boolean tensor rank and tensor decompositions

We now give the definitions of Boolean tensor rank, and Boolean CP and Tucker decompositions. The definitions of Boolean tensor rank and Boolean tensor decompositions 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 4 (*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.,

$$\mathcal{X} = \bigvee_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r. \tag{13}$$

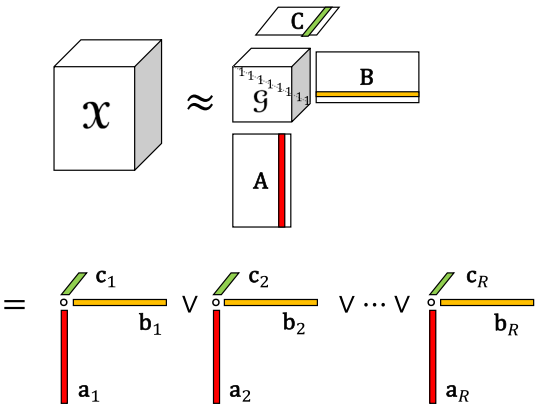


Fig. 1 Rank-R Boolean CP decomposition of a three-way tensor $\mathcal{X}$. $\mathcal{X}$ is decomposed into three binary factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$

Definition 5 (*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

$$\left| \mathcal{X} - \bigvee_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \right|. \tag{14}$$

By replacing the normal matrix product in Eq. (10) with the Boolean matrix product, Boolean CP decomposition can be expressed in a matricized form as follows:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^\top. \end{aligned} \tag{15}$$

Figure 1 illustrates the rank-R Boolean CP decomposition of a three-way tensor $\mathcal{X}$.

Definition 6 (*Boolean Tucker decomposition*) Given a binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$ and the dimensions of a core tensor R_1, R_2, and R_3, find binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a binary core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$ such that they minimize

$$\left| \mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3} \right|. \tag{16}$$

By using Boolean matrix product in place of the normal matrix product in Eq. (12), Boolean Tucker decomposition can be expressed in a matricized form as follows:

$$\begin{aligned} \mathbf{X}_{(1)} &\approx \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top \\ \mathbf{X}_{(2)} &\approx \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top \\ \mathbf{X}_{(3)} &\approx \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top. \end{aligned} \tag{17}$$

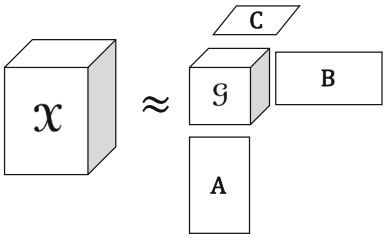


Fig. 2 Rank-R Boolean Tucker decomposition of a three-way tensor $\mathcal{X}$. $\mathcal{X}$ is decomposed into a binary core tensor $\mathcal{G}$, and three binary factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$

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 the maximum number of iterations T.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R}$, $\mathbf{B} \in \mathbb{B}^{J \times R}$, and $\mathbf{C} \in \mathbb{B}^{K \times R}$.

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

Figure 2 illustrates the rank-R Boolean Tucker decomposition of a three-way tensor $\mathcal{X}$.

Algorithm 2: Boolean Tucker Decomposition Framework

Input: A three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, dimensions of a core tensor R_1, R_2, and R_3, and the maximum number of iterations T.
Output: Binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$.

initialize factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$
initialize the core tensor $\mathcal{G}$ such that $\left|\mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3}\right|$ is minimized
for $t \leftarrow 1..T$ **do**
  update $\mathbf{A}$ such that $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top|$ is minimized
  update $\mathbf{B}$ such that $|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top|$ is minimized
  update $\mathbf{C}$ such that $|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top|$ is minimized
  update $\mathcal{G}$ such that $\left|\mathcal{X} - \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} \mathbf{a}_{r_1} \circ \mathbf{b}_{r_2} \circ \mathbf{c}_{r_3}\right|$ is minimized
  if *converged* **then**
    break out of **for** loop
return $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, *and* $\mathcal{G}$

Computing the Boolean CP and Tucker Decompositions The alternating least squares (ALS) algorithm is the "workhorse" approach for normal CP and Tucker decompositions [1]. With a few changes, ALS projection heuristic provides frameworks for computing the Boolean CP and Tucker decompositions as shown in Algorithms 1 and 2.

The framework for Boolean CP decomposition (Algorithm 1) is composed of two parts: first, the initialization of factor matrices (line 1), and second, the iterative update of each factor matrix in turn (lines 3–5). At each step of the iterative update phase, the nth 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 tensor $\mathcal{X}$ and the approximate tensor reconstructed from the factor matrices using Eq. (15), while the other factor matrices are fixed. The framework for Boolean Tucker decomposition (Algorithm 2) is similar to that for Boolean CP decomposition, except for (1) the additional initialization and update of the core tensor in lines 2 and 7, respectively, and (2) the tensor reconstruction in lines 4–6 that involves the core tensor and the Kronecker product between factor matrices (Eq. (17)) The convergence criterion for Algorithms 1 and 2 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 frameworks, Miettinen [3] proposed Boolean CP and Tucker decomposition algorithms named BCP_ALS and BTucker_ALS, respectively. However, since both methods are designed to run on a single machine, their scalability and performance are limited by the computing and memory capacity of a single machine. Also, the initialization scheme used in the two methods 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 and BTucker_ALS cannot scale up to large-scale tensors.

Walk'n'Merge [2] is a different approach for Boolean CP and Tucker factorizations. Representing the tensor as a graph, Walk'n'Merge performs random walks on it to identify dense blocks (which correspond to rank-1 tensors) and merge these blocks to get larger, yet dense blocks; Walk'n'Merge orders and selects blocks based on the minimum description length (MDL) principle for the CP decomposition, and obtains the Tucker decomposition from the returned blocks by merging factors and adjusting the core tensor accordingly again using the MDL principle. As a result, the dimension of a core tensor cannot be controlled with Walk'n'Merge. While Walk'n'Merge is a parallel algorithm, its scalability is still limited for large-scale tensors. 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 Sect. 5.2.

3 Related works

In this section, we review previous approaches for computing Boolean and normal tensor decompositions and present related works on the partitioning of sparse tensors, and distributed computing frameworks.

3.1 Boolean tensor decomposition

Leenen et al. [7] 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 [6], Belohlávek et al. presented a greedy algorithm for Boolean CP decomposition of three-way binary data. In the preliminary version of this work [9], Park et al. proposed a distributed method for Boolean CP factorization running on the Apache Spark framework. Erdős et al. [2] proposed a parallel algorithm called Walk'n'Merge for scalable Boolean CP and Tucker decompositions, which performs random walks to find dense blocks (rank-1 tensors) and obtains final CP and Tucker decompositions from the returned blocks by employing the MDL principle. In [4], Erdős 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. [5] 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 nonzero per row.

3.2 Normal tensor decomposition

Many algorithms have been developed for normal CP and Tucker decompositions.

CP decomposition GigaTensor [12] is the first work for large-scale CP decomposition running on MapReduce. In [13], Jeon et al. proposed SCouT for scalable coupled matrix-tensor factorization. Recently, tensor decomposition methods proposed in [12–15] have been integrated into a multi-purpose tensor mining library, BIGtensor [16]. Beutel et al. [17] proposed FlexiFaCT, a scalable MapReduce algorithm to decompose matrix, tensor, and coupled matrix-tensor using stochastic gradient descent. ParCube [18] is a fast and parallelizable CP decomposition method that produces sparse factors by leveraging random sampling tech-

niques. In [19], Li et al. proposed AdaTM, which adaptively chooses parameters in a model-driven framework for an optimal memoization strategy so as to accelerate the factorization process. Smith et al. [20] and Karlsson et al. [21] both developed alternating least squares (ALS) and coordinate descent (CCD++) methods for parallel CP factorizations; [20] also explored parallel stochastic gradient descent (SGD) method. CDTF [22] provides a scalable tensor factorization method that focuses on nonzero elements of a tensor.

Tucker decomposition De Lathauwer et al. [23] proposed foundational work on N-dimensional Tucker-ALS algorithm. As conventional Tucker-ALS methods suffer from limited scalability, many scalable Tucker methods have been developed. Kolda et al. [24] proposed MET (memory-efficient Tucker), which avoids explicitly constructing intermediate data and maximizes performance while optimally using the available memory. S-HOT [25] further improved the scalability of MET [24] by employing on-the-fly computation and streaming nonzeros of a tensor from the disk. Smith et al. [26] accelerated the factorization process by removing computational redundancies with a compressed data structure. Jeon et al. [14] provided a scalable Tucker decomposition method running on the MapReduce framework. Kaya et al. [27] and Oh et al. [28] designed efficient Tucker algorithms for sparse tensors. Chakaravarthy et al. [29] proposed optimized distributed Tucker decomposition method for dense input tensors.

3.3 Partitioning of sparse tensors

For distributed tensor factorization, it is essential to use efficient partitioning methods so as to maximize parallelism and minimize communication costs between machines. There are various partitioning approaches for decomposing sparse tensors on distributed platforms. DFacTo [30] and CDTF [22] are two systems that perform a coarse-grained partitioning of the input tensor where independent, one-dimensional block partitionings are performed for each tensor mode. With a coarse-grained partitioning, each process has all the nonzeros required for computing its output; thus, the only necessary communication is to exchange updated factor rows at each iteration. However, it has the disadvantage that dense factor matrices need to be sent to all processes in their entirety. Hypergraph partitioning methods [27,31] reduce communication volume via a fine-grained partitioning of the input tensor, in which nonzeros are assigned to processes individually. However, hypergraph partitioning involves expensive preprocessing step, which often takes more time than the actual factorization. Recently, Cartesian (or medium-grained) partitioning methods [32–34] have gained interests due to reduced memory and communication costs, which divide an input tensor into a 3D grid, and factor matrices into corresponding groups of rows. All of the above partitioning methods have been developed for normal tensor factorization, where factor matrices are highly dense and, accordingly, incur a high memory usage and communication overhead. On the other hand, factor matrices in BTF are usually much sparser than the normal factor matrices due to Boolean constraint, and BTF methods can usually process smaller tensors than normal decomposition techniques due to high computational complexity. Considering these characteristics of BTF, DBTF adopts a coarse-grained, vertical partitioning for the unfolded tensor and performs a Cartesian partitioning of the input tensor, which we discuss in Sect. 4.5.

3.4 Distributed computing frameworks

MapReduce [35] 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 [36] is an open-source implementation of MapReduce. Due to the advantages of MapReduce, many data mining tasks [12,37–40] have used Hadoop. However, due to intensive disk I/O, Hadoop is inefficient at executing iterative algorithms [41]. Apache Spark [8,42] 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 [43–47].

4 Proposed method

In this section, we describe DBTF, our proposed method for distributed Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations. 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 the number of operations** Boolean tensor factorization is an NP-hard problem [3] with a high computational cost. How can we minimize the number of operations for factorizing Boolean tensors?
3. **Identify the characteristics of Boolean tensor factorization** In contrast to the normal tensor factorization, Boolean tensor factorization applies Boolean operations

to binary data. How can we utilize 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 (Sect. 4.3).
2. **Exploiting the characteristics of Boolean operation and Boolean tensor factorization** decreases the number of operations to update factor matrices (Sect. 4.4).
3. **Careful partitioning of the workload** facilitates reuse of intermediate results and minimizes data shuffling (Sect. 4.5).

We give an overview of how DBTF updates factor matrices (Sect. 4.1) and a core tensor (Sect. 4.2) and then describe how we address the aforementioned scalability challenges in detail (Sects. 4.3–4.6). After that, we discuss implementation issues (Sect. 4.7) and provide a theoretical analysis of DBTF (Sect. 4.8). While DBTF-CP and DBTF-TK have a lot in common, DBTF-TK deals with some additional challenges. Accordingly, we organize this section such that Sects. 4.1–4.5 describe ideas that apply to both DBTF-CP and DBTF-TK, and Sects. 4.2 and 4.5.2 are dedicated to ideas that apply to DBTF-TK.

4.1 Updating a factor matrix

DBTF is a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations based on the framework described in Algorithms 1 and 2, respectively. The core operation of DBTF-CP and DBTF-TK is updating factor matrices (lines 3–5 in Algorithm 1, and lines 4–6 in Algorithm 2). Since the update steps are similar, we focus on updating the factor matrix $\mathbf{A}$.

DBTF performs a columnwise update row by row. This is done with doubly nested loops, where the outer loop selects a column to update, and the inner loop iterates over the rows of a factor matrix, updating only those entries in the column selected by the outer loop. In other words, DBTF iterates over the rows of factor matrix for R (DBTF-CP) or R_1 (DBTF-TK) column (outer)-iterations in total, updating entries of each row in column c at column-iteration c ($1 \leq c \leq R$ for DBTF-CP, or $1 \leq c \leq R_1$ for DBTF-TK) to the values that result in a smaller reconstruction error. This is a greedy approach that updates each entry to the value that yields a better accuracy while all other entries in the same row are fixed; as a result, it does not consider all combinations of values for factor matrix elements. We also considered an exact approach that explores every possible value assignment, but preliminary tests showed that the greedy heuristic performs very closely to the exact search while being much faster than the exact search which takes exponential time with respect to R or R_1. Figure 3 shows an overview of how DBTF updates a factor matrix. In Fig. 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.

Updating a factor Matrix in DBTF-CP The objective of updating the factor matrix in DBTF-CP is to minimize the difference between the unfolded input tensor $\mathbf{X}_{(1)}$ and the approximate tensor $\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$. Let c refer to the column to be updated. Then, DBTF-CP computes $|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top|$ for each of the possible values for the entries in column c (i.e., $\mathbf{a}_{:c}$) and updates the column c to the set of values that yield the smallest difference. In order to calculate the difference at row-iteration i, $[\mathbf{X}_{(1)}]_{i:}$ (gray rectangle in $\mathbf{X}_{(1)}$ of Fig. 3) is compared against $[\mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top]_{i:} = \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ (Fig. 3a). Then, an entry in a_{ic} is updated to the value that gives a smaller difference, i.e., $\left|[\mathbf{X}_{(1)}]_{i:} - \mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top\right|$.

Updating a factor matrix in DBTF-TK DBTF-TK updates the factor matrix such that the difference between $\mathbf{X}_{(1)}$ and $\mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ is minimized. DBTF-TK calculates the difference at row-iteration i by comparing $[\mathbf{X}_{(1)}]_{i:}$ against $[\mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top]_{i:} = [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ (Fig. 3b), and updates an entry in a_{ic} to the value resulting in a smaller difference, i.e., $\left|[\mathbf{X}_{(1)}]_{i:} - [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top\right|$.

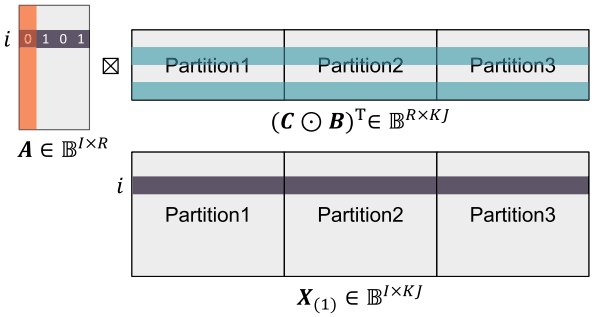
Lemma 1 *$\mathbf{a}_{i:} \boxtimes \mathbf{B}$ is the same as selecting rows of $\mathbf{B}$ that correspond to the indices of nonzeros of $\mathbf{a}_{i:}$, and performing a Boolean summation of those rows.*

Proof This follows directly from the definition of Boolean matrix product $\boxtimes$ (Eq. (7)). □

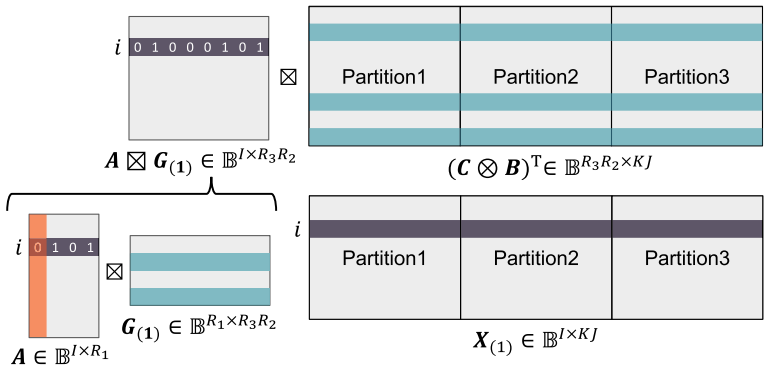
Consider Fig. 3a as an example: Since $\mathbf{a}_{i:}$ is 0101 (gray rectangle in $\mathbf{A}$), $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ is identical to the Boolean summation of the second and fourth rows of $(\mathbf{C} \odot \mathbf{B})^\top$ (blue rectangles). Similarly, in Fig. 3b, $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ is the same as the Boolean summation of the second, sixth, and eighth rows of $(\mathbf{C} \otimes \mathbf{B})^\top$ (blue rectangles) as $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ is 01000101.

Note that an update of the ith row of $\mathbf{A}$ does not depend on those of its other rows since $\mathbf{a}_{i:} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top$ or $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top$ needs to be compared only with $[\mathbf{X}_{(1)}]_{i:}$. Therefore, the determination of whether to update an entry of some row in $\mathbf{a}_{:c}$ to 0 or 1 can be made independently of the decisions for entries in other rows. Also, notice in Fig. 3b that while it is factor matrix $\mathbf{A}$ that DBTF-TK tries to update, it is not the rows in $\mathbf{A}$ that determine which rows in $(\mathbf{C} \otimes \mathbf{B})^\top$ are to be summed as in Fig. 3a, but those in the intermediate matrix $\mathbf{A} \boxtimes \mathbf{G}_{(1)}$.

Depending on the distribution of nonzeros in the input tensor, and how factor matrices and a core tensor have been



(a) Updating a factor matrix for Boolean CP factorization by DBTF-CP.



(b) Updating a factor matrix for Boolean Tucker factorization by DBTF-TK.

Fig. 3 An overview of updating a factor matrix for **a** Boolean CP factorization by DBTF-CP, and **b** Boolean Tucker factorization by DBTF-TK. DBTF performs a columnwise update row by row. DBTF iterates over the rows of factor matrix for R (DBTF-CP) or R_1 (DBTF-TK) column (outer)-iterations in total, updating entries of each row in column c at column-iteration c ($1 \leq c \leq R$ for DBTF-CP, or $1 \leq c \leq R_1$ for DBTF-TK) to the values that result in a smaller reconstruction error. 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 $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ are the rows that are Boolean summed to be compared against the ith row of $\mathbf{X}_{(1)}$ (gray rectangle in $\mathbf{X}_{(1)}$). Vertical blocks in $(\mathbf{C} \odot \mathbf{B})^\top$, $(\mathbf{C} \otimes \mathbf{B})^\top$, and $\mathbf{X}_{(1)}$ represent partitions of the data (see Sect. 4.5 for details on partitioning) (color figure online)

initialized and updated, the factor matrix currently being updated may be updated to contain only zeros. When this happens, the intermediate matrix constructed with a Khatri–Rao (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$) or a Kronecker product (e.g., $(\mathbf{C} \otimes \mathbf{B})^\top$) at the following iteration will consist of only zeros, and as a result, the difference between the approximate tensor and the input tensor will be always the same, regardless of how the factor matrix is updated. We handle this issue by providing the ability to upper bound the maximum percentage of zeros in the column being updated. When the percentage of zeros in the current column exceeds the given threshold, DBTF finds values different from the current assignments for a subset of rows, which will make the sparsity of the current column become less than the upper bound with the smallest increase in error, and updates those rows accordingly.

4.2 Updating a core tensor

Boolean Tucker factorization involves an additional task of updating a core tensor $\mathcal{G}$. How DBTF-TK updates a core tensor is based on BTucker_ALS [3]. Below we describe the main observations used by DBTF-TK and BTucker_ALS for updating $\mathcal{G}$ and explain how DBTF-TK further reduces the amount of computation.

Given the definition of Boolean Tucker decomposition (Eq. (16)), an (i, j, k)th element of an approximate tensor $\tilde{\mathcal{X}}$ is computed as follows:

$$\tilde{x}_{ijk} = \bigvee_{r_1=1}^{R_1} \bigvee_{r_2=1}^{R_2} \bigvee_{r_3=1}^{R_3} g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3}. \tag{18}$$

That is, every element of $\mathcal{G}$ is involved with the computation of $\tilde{x}_{ijk}$, and thus, flipping an element in $\mathcal{G}$ can affect the entire $\tilde{\mathcal{X}}$. However, we observe that the value of $g_{r_1 r_2 r_3}$ does not affect the product $g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3}$ if $a_{i r_1} b_{j r_2} c_{k r_3}$ is 0. Therefore, only those (i, j, k)s for which $a_{i r_1} b_{j r_2} c_{k r_3} \neq 0$ are considered in DBTF-TK and BTucker_ALS. We also notice that, as a result of Boolean sum, if there exists some (r_1, r_2, r_3) such that $g_{r_1 r_2 r_3} a_{i r_1} b_{j r_2} c_{k r_3} = 1$, then $\tilde{x}_{ijk} = 1$ regardless of values of other elements in $\mathcal{G}$.

Based on these observations, both methods compute the partial gain of flipping the (r_1, r_2, r_3)th element for which there exists some (i, j, k) such that $a_{ir_1} b_{jr_2} c_{kr_3} = 1$, and update those elements having a positive gain.

- If $g_{r_1 r_2 r_3}$ and $\tilde{x}_{ijk}$ are both 0, then there exists no $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ such that $g_{\alpha\beta\gamma} a_{i\alpha} b_{j\beta} c_{k\gamma} = 1$. If $x_{ijk} = 1$ in this case, setting $g_{r_1 r_2 r_3}$ to 1 results in a partial gain, since $g_{r_1 r_2 r_3} a_{ir_1} b_{jr_2} c_{kr_3}$ becomes 1, and $\tilde{x}_{ijk} = x_{ijk} = 1$.
- If $g_{r_1 r_2 r_3}$ is 1, $\tilde{x}_{ijk}$ is guaranteed to be 1. However, flipping $g_{r_1 r_2 r_3}$ back to 0 does not necessarily lead to a partial gain since there might be other $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ such that $g_{\alpha\beta\gamma} a_{i\alpha} b_{j\beta} c_{k\gamma} = 1$. So in this case, under the condition that no such (α, β, γ) exists and x_{ijk} is 0, setting $g_{r_1 r_2 r_3}$ to 0 leads to a partial gain.

We further reduce the amount of computation by utilizing vectors s_I, s_J, and s_K, which contain the rowwise sum of entries in factor matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$ that are in those columns selected by entries in $\mathcal{G}$. Let us assume that $a_{ir_1} b_{jr_2} c_{kr_3} = 1$ for some (i, j, k) and (r_1, r_2, r_3). First, when $g_{r_1 r_2 r_3} = 0$ and $x_{ijk} = 1$, we need to know whether $\tilde{x}_{ijk}$ is 0 or not in order to compute the partial gain. We observe that $\tilde{x}_{ijk} = 0$ if at least one of $s_I(i), s_J(j)$, and $s_K(k)$ is zero, since when this condition is satisfied, $a_{i\alpha} b_{j\beta} c_{k\gamma} = 0$ for any (α, β, γ) in $\mathcal{G}$. Second, if $g_{r_1 r_2 r_3} = 1$, $\tilde{x}_{ijk}$ is equal to 1. When $x_{ijk} = 1$ in this case, in order to compute the partial gain, we need to know whether there exists some $(\alpha, \beta, \gamma) \neq (r_1, r_2, r_3)$ that also contributes to $\tilde{x}_{ijk} = 1$. We note that if all of $s_I(i), s_J(j)$, and $s_K(k)$ are equal to one, then $g_{r_1 r_2 r_3}$ is the only element in $\mathcal{G}$ that turns on $\tilde{x}_{ijk}$, since otherwise, there exists at least one other element in $\mathcal{G}$ also contributing to $\tilde{x}_{ijk}$, which is impossible given that $s_I(i) = s_J(j) = s_K(k) = 1$. In both cases, vectors s_I, s_J, and s_K help us avoid visiting elements in $\mathcal{G}$ repeatedly, and enable DBTF-TK to skip the current (i, j, k) and the following ones for which no partial gain can be obtained.

While the above ideas allow the update of a core tensor $\mathcal{G}$, updating $\mathcal{G}$ in a distributed environment poses a challenge of how to distribute the workload among machines, which we describe in Sect. 4.5.2.

4.3 Distributed generation and minimal transfer of intermediate data

The first challenge for performing Boolean tensor factorization in a distributed manner is how to generate and distribute the intermediate data efficiently. In particular, updating a factor matrix involves the following types of intermediate data: (1) a Khatri–Rao product of two factor matrices (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$), (2) a Kronecker product of two factor matrices (e.g., $(\mathbf{C} \otimes \mathbf{B})^\top$), and (3) an unfolded tensor (e.g., $\mathbf{X}_{(1)}$).

Khatri–Rao and Kronecker products A naive method for processing the Khatri–Rao and Kronecker products is to construct the entire 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 [5], performing the entire Khatri–Rao or Kronecker product is still an expensive operation. Also, since one of the two matrices involved in the product is always updated in the previous update procedure (Algorithms 1 and 2), prior Khatri–Rao or Kronecker 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,

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{b}_{:1} & a_{12}\mathbf{b}_{:2} & \cdots & a_{1R}\mathbf{b}_{:R} \\ a_{21}\mathbf{b}_{:1} & a_{22}\mathbf{b}_{:2} & \cdots & a_{2R}\mathbf{b}_{:R} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{b}_{:1} & a_{I2}\mathbf{b}_{:2} & \cdots & a_{IR}\mathbf{b}_{:R} \end{bmatrix}, \tag{19}$$

and that of Kronecker product (Eq. (2)). We notice from Eqs. (2) and (19) that a specific range of rows of Khatri–Rao or Kronecker 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 product.

Unfolded tensor While the Khatri–Rao or Kronecker products are computed iteratively, matricizations of an input tensor need to be performed only once. However, in contrast to the Khatri–Rao and Kronecker products, we cannot avoid shuffling the entire unfolded tensor as we have no characteristics to exploit as in the case of Khatri–Rao or Kronecker 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 afterward. We do vertical partitioning of the Khatri–Rao and Kronecker products and unfolded tensors as shown in Fig. 3 (see Sect. 4.5 for more details on the partitioning of unfolded tensors).

4.4 Exploiting the characteristics of Boolean operation and Boolean tensor factorization

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

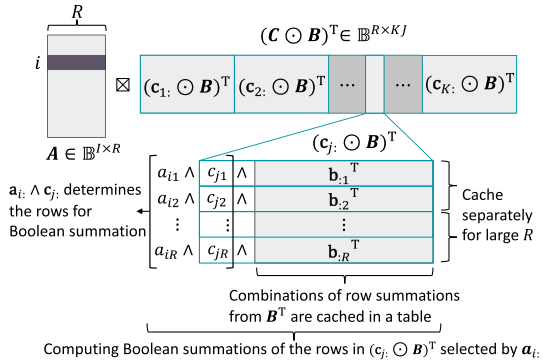


Fig. 4 DBTF-CP reduces intermediate operations by exploiting the characteristics of Boolean CP factorization. Blue rectangles in $(\mathbf{C} \odot \mathbf{B})^\top$ correspond to K vector-matrix Khatri–Rao products, among which $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$ is shown in detail. $\mathbf{B}^\top$ is the target for row summation. A Boolean vector $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ determines which rows in $\mathbf{B}^\top$ are to be summed to compute the row summation of $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$. Combinations of the row summations of $\mathbf{B}^\top$ are cached. For large R, rows of $\mathbf{B}^\top$ are split into multiple, smaller groups, each of which is cached separately (color figure online)

4.4.1 Problem

Given our procedure to update factor matrices (Sect. 4.1), the two most frequently performed tasks are (1) computing the Boolean sums of selected rows of $(\mathbf{C} \odot \mathbf{B})^\top$ (CP factorization) or $(\mathbf{C} \otimes \mathbf{B})^\top$ (Tucker factorization), 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, the first task takes $O(RI^2)$ or $O(R^2I^2)$ time (for CP and Tucker factorizations, respectively), and the second task takes $O(I^2)$ time. Since we compute the errors for both cases of when each factor matrix entry is set to 0 and 1, each task 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 task $6TRI$ times in total. Due to high computational costs and a large number of repetitions, it is crucial to minimize the number of intermediate operations involved with these tasks.

4.4.2 Our solution

Overview We start with the following observations:

- By Lemma 1, DBTF computes the Boolean sum of selected rows of $(\mathbf{C} \odot \mathbf{B})^\top$ (DBTF-CP), or $(\mathbf{C} \otimes \mathbf{B})^\top$ (DBTF-TK). This amounts to performing a specific set of operations repeatedly, which we describe below.
- Khatri–Rao and Kronecker products can be expressed by vector-matrix (VM) Khatri–Rao and Kronecker products, respectively (Eqs. (3) and (5)).
- Given factor matrices of size I-by-R, there are 2^R and 2^{R^2} combinations of selecting rows from $(\mathbf{C} \odot \mathbf{B})^\top \in \mathbb{B}^{R \times I^2}$ and $(\mathbf{C} \otimes \mathbf{B})^\top \in \mathbb{B}^{R^2 \times I^2}$, respectively.

Our main idea is to exploit the characteristics of Boolean operation and Boolean tensor factorization as summarized in the above observations to reduce the number of intermediate steps to perform Boolean row summations. Figures 4 and 5 present an overview of our idea for Boolean CP and Tucker factorizations. We note that according to Eq. (4),

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

Similarly, we notice that by Eq. (2),

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

Blue rectangles in $(\mathbf{C} \odot \mathbf{B})^\top$ and $(\mathbf{C} \otimes \mathbf{B})^\top$ (Figs. 4, 5) correspond to K VM Khatri–Rao products, $[(\mathbf{c}_{1:} \odot \mathbf{B})^\top, \ldots, (\mathbf{c}_{K:} \odot \mathbf{B})^\top]$, and K VM Kronecker products, $[(\mathbf{c}_{1:} \otimes \mathbf{B})^\top, \ldots, (\mathbf{c}_{K:} \otimes \mathbf{B})^\top]$, respectively. Since a row of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ is made up of a sequence of K corresponding rows of VM Khatri–Rao or VM Kronecker products, the Boolean sum of the selected rows of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$ can be constructed by summing up the same set of rows in each VM Khatri–Rao or VM Kronecker product, and concatenating the resulting rows into a single row.

Selecting rows of $\mathbf{B}^\top$ in DBTF-CP Assuming that the row $\mathbf{a}_{i:}$ is being updated as in Fig. 4, we observe that computing Boolean row summations of each $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$ amounts to summing up the rows in $\mathbf{B}^\top$ that are selected by the next two conditions. First, we choose all those rows of $\mathbf{B}^\top$ whose corresponding entries in $\mathbf{c}_{j:}$ are 1. Since all other rows are empty vectors by the definition of Khatri–Rao product (Eq. (4)), they can be ignored in computing Boolean row summations. Second, we pick the set of rows from each $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$ 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 $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$.

Selecting rows of $\mathbf{B}^\top$ in DBTF-TK Assuming that the row $\mathbf{a}_{i:}$ is being updated, we can compute Boolean row summations of each $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ by employing an approach similar to that used for Boolean CP factorization, which is to sum up those rows in $[\mathbf{B}\ \mathbf{B}\ \ldots\ \mathbf{B}]^\top \in \mathbb{B}^{R_3R_2 \times J}$ that are selected by the nonzeros of $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ and $\mathbf{c}_{j:}$, as depicted in Fig. 5. While straightforward, this approach is not efficient as in Boolean CP factorization as the number of rows of the intermediate $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ is R_3 times that of $(\mathbf{c}_{j:} \odot \mathbf{B})^\top$. Furthermore, we observe in Fig. 5 that $\mathbf{B}^\top$ is repeatedly involved in constructing $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$; therefore, if we know which rows of $\mathbf{B}^\top$ are to be selected by $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i:}$ and $\mathbf{c}_{j:}$, we can obtain

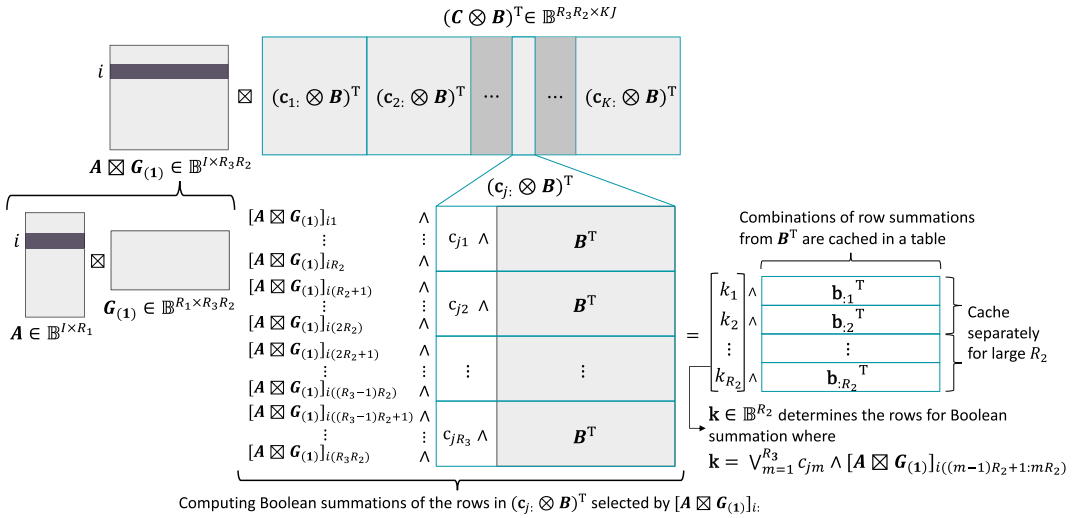


Fig. 5 DBTF-TK reduces intermediate operations by exploiting the characteristics of Boolean Tucker factorization. Blue rectangles in $(\mathbf{C} \otimes \mathbf{B})^\top$ correspond to K vector-matrix Kronecker products, among which $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$ is shown in detail. $\mathbf{B}^\top$ is the target for row summation. A Boolean vector$\mathbf{k} = \bigvee_{m=1}^{R_3} c_{jm} \wedge [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$ determines which rows in $\mathbf{B}^\top$ are to be summed to compute the row summation of $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$. Combinations of the row summations of $\mathbf{B}^\top$ are cached. For large R_2, rows of $\mathbf{B}^\top$ are split into multiple, smaller groups, each of which is cached separately (color figure online)

the Boolean row summation immediately from $\mathbf{B}^\top$, without going over $\mathbf{B}^\top$ R_3 times. The value of Boolean AND ($\wedge$) between c_{jm} and $[\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$ determines which rows are to be summed among the rows of the mth $\mathbf{B})^\top$ in Fig. 5. Thus, the Boolean OR ($\vee$) of all these Boolean ANDs determines which rows need to be summed together to obtain the Boolean row summation for $(\mathbf{c}_{j:} \otimes \mathbf{B})^\top$.

Caching In computing a row summation of $(\mathbf{C} \odot \mathbf{B})^\top$ or $(\mathbf{C} \otimes \mathbf{B})^\top$, we repeatedly sum a subset of rows in $\mathbf{B}^\top$ selected by the aforementioned conditions for each VM Khatri–Rao or Kronecker product. Then, if we can reuse row summation results, we can avoid summing up the same set of rows again and again. DBTF precalculates combinations of row summations of $\mathbf{B}^\top$ and caches the results in a table in memory. This table maps a specific subset of selected rows in $\mathbf{B}^\top$ to its Boolean summation result. In summary, we use the followings as a key to this cache table:

- $\mathbf{a}_{i:} \wedge \mathbf{c}_{j:}$ for Boolean CP factorization
- $\bigvee_{m=1}^{R_3} c_{jm} \wedge [\mathbf{A} \boxtimes \mathbf{G}_{(1)}]_{i((m-1)R_2+1:mR_2)}$ for Boolean Tucker factorization

An issue related to this approach is that the space required for the table increases exponentially with the rank size. Thus, when R becomes larger than a threshold value V, we divide rows evenly into $\lceil R/V \rceil$ smaller groups, construct smaller tables for each group, and then perform additional Boolean summation of rows that we obtain 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 $\mathbf{b}_{:1}{}^\top, \ldots, \mathbf{b}_{:9}^\top$, and the second one storing those of $\mathbf{b}_{:10}{}^\top, \ldots, \mathbf{b}_{:18}^\top$. 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.

In addition to the cache table containing the row summation results of $\mathbf{B}^\top$, we build another cache table for Boolean Tucker factorization, which maps a set of rows of an unfolded core tensor (e.g., $\mathbf{G}_{(1)}$) to its Boolean summation result. Note that, in contrast to the case of Boolean CP factorization, the row $\mathbf{a}_{i:}$ is not directly used for computing a cache key in Boolean Tucker factorization. Instead, $\mathbf{a}_{i:}$ determines the set of rows of $\mathbf{G}_{(1)}$ that are to be summed, and the resulting row summation, $\mathbf{a}_{i:} \boxtimes \mathbf{G}_{(1)}$, is then used for constructing the cache key. In order to avoid summing up the same set of rows in $\mathbf{G}_{(1)}$ repeatedly, DBTF-TK also precomputes the combinations of row summations of $\mathbf{G}_{(1)}$ and stores them in an in-memory table. For large R, this additional table is also split into smaller ones in the same way as discussed above.

Note that the benefit of caching depends on a few factors such as the density of factors and a core tensor, the threshold value V, and the upper bound on the maximum percentage of zeros in columns of a factor matrix (Sect. 4.1). When factors and a core tensor are updated to be sparse, it is advisable to limit V to some small values such that we calculate combi-

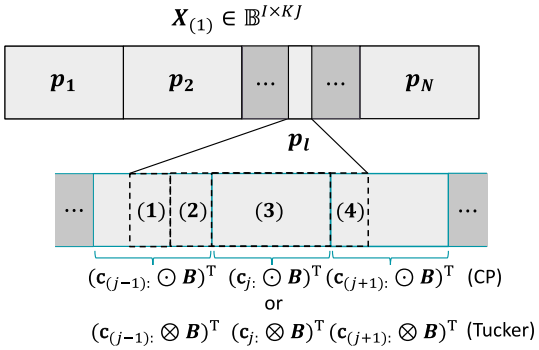


Fig. 6 An overview of partitioning. DBTF partitions the unfolded input tensor vertically into a total of N partitions $p_1, p_2, \ldots, p_N$, among which the lth partition p_l is shown in detail. A partition is further divided into "blocks" (rectangles in dashed lines) by the vertical boundaries between the underlying vector-matrix (VM) Khatri–Rao (CP factorization) or VM Kronecker (Tucker factorization) products, which correspond to blue rectangles. Numbers in p_l refer to the types of blocks a partition can be split into

nations in a small amount of time, while avoiding computing too many combinations that are less likely to be used. When factorizing a dense tensor, it is recommended to try a higher value for V to benefit more from caching.

4.5 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. In the following subsections, we describe how DBTF-CP and DBTF-TK partition unfolded tensors (Sect. 4.5.1), and how DBTF-TK partitions an input tensor (Sect. 4.5.2).

4.5.1 Unfolded tensors

As introduced in Sect. 4.3, 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 dimensionality I, which is usually smaller than the product of dimensionalities KJ. Thus, the maximum number of partitions supported by horizontal partitioning is normally smaller than that by vertical partitioning, which could lower the level of parallelism.

Since the workloads are vertically partitioned, each partition computes an error only for the vertically split 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.

DBTF further splits the vertical partitions of an unfolded tensor in a computation-friendly manner. By "computation-friendly," we mean structuring the partitions in such a way that facilitates an efficient computation of row summation results as discussed in Sect. 4.4. This is crucial since the number of operations performed directly affects the performance of DBTF. The target for row summation in DBTF is $\mathbf{B}^\top$ as shown in Figs. 4 and 5. However, the horizontal length of each partition is not always the same as or a multiple of that of $\mathbf{B}^\top$. Depending on the number of partitions, and the size of $\mathbf{C}$ and $\mathbf{B}$, a partition may cross the vertical boundaries of multiple VM Khatri–Rao or Kronecker products, or may be shorter than one VM Khatri–Rao or Kronecker product.

Figure 6 presents an overview of our idea for computation-friendly partitioning in DBTF. DBTF partitions the unfolded input tensor into a total of N partitions $p_1, p_2, \ldots, p_N$, among which the lth partition p_l is shown in detail. A partition is further divided into "blocks" (rectangles in dashed lines) by the vertical boundaries between the underlying Khatri–Rao (CP factorization) or Kronecker (Tucker factorization) products, which correspond to blue rectangles. Numbers in p_l refer to the types of blocks a partition can be split into. Since the target for row summation is $\mathbf{B}^\top$, with this organization, each block of a partition can efficiently obtain its row summation results.

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 Fig. 6. If the horizontal length of a partition is smaller than or equal to that of a single Khatri–Rao or Kronecker product, it can consist of up to two blocks. When the partition does not cross the vertical boundary between Khatri–Rao or Kronecker 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 vertical boundary between products, it consists of two blocks of types (2) and (4).

If the horizontal length of a partition is larger than that of a single Khatri–Rao or Kronecker 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, in all cases, a partition can have at most three types of blocks. □

An issue with respect to the use of caching is that the horizontal length of blocks of types (1), (2), and (4) is smaller than that of a single Khatri–Rao or Kronecker 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.

4.5.2 Input tensor

DBTF-TK updates a core tensor $\mathcal{G}$ at each iteration (Algorithm 2). In DBTF-TK, an input tensor $\mathcal{X}$ is necessary for updating $\mathcal{G}$: As described in Sect. 4.2, the way DBTF-TK updates an element $g_{r_1r_2r_3}$ of a core tensor requires accessing an input tensor element x_{ijk} for all i, j, and k for which $a_{ir_1}b_{jr_2}c_{kr_3} = 1$.

For distributed computation, the workload of updating a core tensor $\mathcal{G}$ needs to be partitioned across the cluster. Considering that $\mathcal{G}$ is updated entry by entry in DBTF-TK, and updating each element of $\mathcal{G}$ requires accessing an input tensor $\mathcal{X}$ entry by entry, we take into account two different workload partitioning approaches: (1) partitioning the core tensor $\mathcal{G}$ and (2) partitioning the input tensor $\mathcal{X}$. Partitioning the core tensor indicates that entries in different partitions of $\mathcal{G}$ are updated concurrently. However, in DBTF-TK, updating an entry (α, β, γ) in $\mathcal{G}$ involves accessing $(\alpha', \beta', \gamma') \neq (\alpha, \beta, \gamma)$. As each partition updates a different part of $\mathcal{G}$, different partitions may have different views of $\mathcal{G}$, which would result in an incorrect update. Thus, it is not possible to update different entries in $\mathcal{G}$ concurrently. By partitioning the input tensor, on the other hand, only one entry of $\mathcal{G}$ is updated at a time, while entries in different partitions of an input tensor $\mathcal{X}$ are processed in parallel. In this way, all partitions share the global view of the core tensor, and each partition computes the partial gain that can be obtained by flipping an entry (α, β, γ) in $\mathcal{G}$ with regard to the entries of an input tensor assigned to it.

DBTF-TK partitions the input tensor $\mathcal{X}$ into a total of N non-overlapping subtensors ${}_p\mathcal{X}_1, {}_p\mathcal{X}_1, \ldots, {}_p\mathcal{X}_N$ such that they satisfy the following conditions:

Algorithm 3: DBTF-CP Algorithm

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I\times J\times K}$, rank R, the maximum number of iterations T, the number of sets of initial factor matrices L, the number of partitions N, and a threshold value V to limit the size of a single cache table.
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 ${}_p\mathbf{X}_{(1)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(1)}, N)$
2 ${}_p\mathbf{X}_{(2)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(2)}, N)$
3 ${}_p\mathbf{X}_{(3)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(3)}, N)$
4 **for** $t \leftarrow 1, \ldots, T$ **do**
5 **if** $t = 1$ **then**
6 initialize L sets of factor matrices $(\mathbf{A}_1, \mathbf{B}_1, \mathbf{C}_1), \ldots, (\mathbf{A}_L, \mathbf{B}_L, \mathbf{C}_L)$ randomly where $\mathbf{A}_i \in \mathbb{B}^{I\times R}$, $\mathbf{B}_i \in \mathbb{B}^{J\times R}$, and $\mathbf{C}_i \in \mathbb{B}^{K\times R}$ for $i = 1, 2, \ldots, L$
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`$(\mathbf{A}, \mathbf{B}, \mathbf{C})$
 /* minimize $\left|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes (\mathbf{C} \odot \mathbf{B})^\top\right|$ */
15 $\mathbf{A} \leftarrow$ `UpdateFactorCP`$({}_p\mathbf{X}_{(1)}, \mathbf{A}, \mathbf{C}, \mathbf{B}, V)$
 /* minimize $\left|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes (\mathbf{C} \odot \mathbf{A})^\top\right|$ */
16 $\mathbf{B} \leftarrow$ `UpdateFactorCP`$({}_p\mathbf{X}_{(2)}, \mathbf{B}, \mathbf{C}, \mathbf{A}, V)$
 /* minimize $\left|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes (\mathbf{B} \odot \mathbf{A})^\top\right|$ */
17 $\mathbf{C} \leftarrow$ `UpdateFactorCP`$({}_p\mathbf{X}_{(3)}, \mathbf{C}, \mathbf{B}, \mathbf{A}, V)$
18 **return** $\mathbf{A}, \mathbf{B}, \mathbf{C}$

1. ${}_p\mathcal{X}_t$ is associated with three ranges, I_t, J_t, and K_t, such that $|I_i| \times |J_i| \times |K_i| \approx |I_j| \times |J_j| \times |K_j|$ for all $i, j \in [1 .. N]$.
2. ${}_p\mathcal{X}_t$ contains all $x_{ijk} \in \mathcal{X}$ for $i \in I_t$, $j \in J_t$, and $k \in K_t$.
3. $\bigcup_{t=1}^{N} {}_p\mathcal{X}_t = \mathcal{X}$, and ${}_p\mathcal{X}_i \cap {}_p\mathcal{X}_j = \emptyset$ for all $i, j \in [1 .. N]$ $(i \neq j)$.

With the input tensor partitioned as above, the parallel updates of a core tensor $\mathcal{G}$ in N partitions are synchronized on the update of each entry in $\mathcal{G}$.

4.6 Putting things together

In this section, we present algorithms for DBTF and provide a brief description of their relationships: DBTF-CP is given in Algorithms 3–7, and DBTF-TK is presented in Algorithms 4 and 6–11. The "**distributed**" keyword in the algorithm indicates that the marked section is performed in a fully distributed manner. We also briefly summarize what data are transferred across the network.

Algorithm 4: `PartitionUnfoldedTensor`

Input: an unfolded binary tensor $\mathbf{X} \in \mathbb{B}^{P \times Q}$, and the 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 $\forall i \in \{1, \ldots, N\}$, $p_i \in \mathbb{B}^{P \times H}$ where $\left\lfloor \frac{Q}{N} \right\rfloor \leq H \leq \left\lceil \frac{Q}{N} \right\rceil$
2 ${}_p\mathbf{X} \leftarrow [p_1\ p_2\ \ldots\ p_N]$
3 **D: foreach** $p' \in {}_p\mathbf{X}$ **do**
4 further split p' into a set of blocks divided by the boundaries of underlying pointwise vector-matrix products as depicted in Fig. 6 (see Sect. 4.5)
5 **D:** cache ${}_p\mathbf{X}$ across machines
6 **return** ${}_p\mathbf{X}$

4.6.1 Partitioning

DBTF-CP and DBTF-TK first partition the unfolded input tensors (lines 1–3 in Algorithms 3 and 8): Each unfolded tensor is vertically partitioned and then cached across machines (Algorithm 4). In addition to the unfolded input tensor, DBTF-TK also partitions and caches the input tensor (Algorithm 9).

4.6.2 Updating factor matrices and a core tensor

DBTF-CP and DBTF-TK initialize L sets of factor matrices (line 6 in Algorithm 3 and line 7 in Algorithm 8, respectively). Instead of initializing a single set of factor matrices, DBTF allows initializing multiple sets as better initial factor matrices could lead to more accurate factorization. DBTF-CP 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 3). After initializing factor matrices, DBTF-TK also prepares core tensors for each set of factor matrices (line 9 in Algorithm 8) and finds the set of factor matrices and a core tensor with the smallest error (lines 10–11 in Algorithm 8). In each iteration, factor matrices are updated one at a time, while the other two are fixed (lines 15–17 in Algorithm 3 and lines 19–21 in Algorithm 8). In DBTF-TK, the core tensor is also updated before factor matrices are updated (line 13 in Algorithm 8).

Updating a factor matrix The procedures for updating a factor matrix are shown in Algorithm 5 (DBTF-CP) and Algorithm 10 (DBTF-TK). Note that their core operations—computing a Boolean row summation and its error—are performed in a fully distributed manner (lines 7–9 in Algorithm 5, and lines 7–11 in Algorithm 10). DBTF caches combinations of Boolean row summations of a factor matrix (Algorithm 6) at the beginning of `UpdateFactorCP` and `UpdateFactorTK` to avoid repeatedly computing them. DBTF-TK additionally caches the combinations of row summation results of an unfolded core tensor (line 2 in Algorithm 10). DBTF-CP and DBTF-TK fetch the cached Boolean summation results in an almost identical manner, except that they use different cache keys (line 7 in Algorithm 5, and line 9 in Algorithm 10). DBTF collects errors computed across machines and updates the current column DBTF is visiting (lines 10–14 in Algorithm 5, and lines 12–16 in Algorithm 10). 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.

Updating a core tensor The procedure for updating a core tensor is given in Algorithm 11. DBTF-TK computes the partial gain of flipping an element of a core tensor $\mathcal{G}$ in a fully distributed fashion (lines 4–25 in Algorithm 11) and determines its value using the sum of collected gains (lines 26–27 in Algorithm 11).

4.6.3 Network transfer

In DBTF-CP and DBTF-TK, the following 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. DBTF-TK sends out further data: Partitions of an input tensor are distributed once in the beginning, a core tensor is transferred when factor matrices are updated, and the row-wise sum of entries in factor matrices is distributed when a core tensor is updated.

In both DBTF-CP and DBTF-TK, machines send intermediate errors back to the driver node for the update of columns of a factor matrix. In DBTF-TK, each machine additionally sends the partial gain back to the driver node in order to update a core tensor.

4.7 Implementation

In this section, we discuss practical issues pertaining to the implementation of DBTF on Spark. We use sparse representation for tensors and matrices, storing only nonzero elements, except for those factor matrices to which we apply Boolean AND operation to compute a cache key, which we represent as an array of *BitSet*. An input tensor is loaded as an RDD (*resilient distributed datasets*) [8] 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 nonzeros by partition ID and organizes them into blocks. In Tucker factorization, we similarly use *map* and *groupByKey* operations, to divide the input tensor RDD into partitions, in which nonzeros are organized as a set, since DBTF-TK queries the existence of tensor

Algorithm 5: `UpdateFactorCP`

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 the Khatri–Rao product), and $\mathbf{M}_s \in \mathbb{B}^{S \times R}$ (second matrix for the Khatri–Rao product), a threshold value V to limit the size of a single cache table, and the maximum percentage Z of zeros in the column being updated.

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

`AugmentPartitionWithRowSummations`$({}_p\mathbf{X}, \mathbf{M}_s, V)$
`/* iterate over columns and rows of` $\mathbf{A}$ `*/`
for *column iter* $c \leftarrow 1 \ldots R$ **do**
  for *row* $r \leftarrow 1 \ldots P$ **do**
    for $a_{rc} \leftarrow 0, 1$ **do**
      distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
        foreach *block* $b \in p'$ **do**
          compute the cache key $\mathbf{k} \leftarrow \mathbf{a}_{r:} \wedge [\mathbf{M}_f]_{i:}$ where i is the row index of $\mathbf{M}_f$ such that block b is within the vertical boundaries of underlying $([\mathbf{M}_f]_{i:} \odot \mathbf{M}_s)^\top$
          $\mathbf{v} \leftarrow$ using $\mathbf{k}$, fetch the cached Boolean row summation that corresponds to $\mathbf{a}_{r:} \boxtimes ([\mathbf{M}_f]_{i:} \odot \mathbf{M}_s)^\top$
          compute the error between the fetched row $\mathbf{v}$ and the corresponding part of ${}_p\mathbf{x}_{r:}$
  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)
  for *row* $r \leftarrow 1 \ldots P$ **do** `/* update` $\mathbf{a}_{:c}$ `*/`
    update a_{rc} to the value that yields a smaller error (i.e., $\left|\mathbf{x}_{r:} - \mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^\top\right|$)
  if *the percentage of zeros in* $\mathbf{a}_{:c} > Z$ **then**
    find new values for a subset of rows which will make $\mathbf{a}_{:c}$ to obey Z with the smallest increase in error, and update those rows accordingly.
return $\mathbf{A}$

Algorithm 6: `AugmentPartitionWithRowSummations`

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.

distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
  $\mathbf{T}_i \leftarrow$ `GenerateRowSummations`$(\mathbf{M}_c, V)$
  foreach *block* $b \in p'$ **do**
    if block b is of the type (1), (2), or (4) as shown in Fig. 6, vertically slice $\mathbf{T}_i$ such that the sliced one corresponds to block b
    cache (the sliced) $\mathbf{T}_i$ if not cached, and augment partition p' with it

Algorithm 7: `GenerateRowSummations`

Input: 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.

Output: a table $\mathbf{T}_m$ that contains mappings from a set of rows in $\mathbf{M}_c$ to its summation result

$\mathbf{T}_m \leftarrow$ 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 generate combinations of row summations from each one separately)
return $\mathbf{T}_m$

Algorithm 8: DBTF-TK Algorithm

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, dimensions of a core tensor R_1, R_2, and R_3, the maximum number of iterations T, the number of sets of initial factor matrices L, the number of partitions N, and a threshold value V to limit the size of a single cache table.

Output: binary factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$, and a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$.

${}_p\mathbf{X}_{(1)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(1)}, N)$
${}_p\mathbf{X}_{(2)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(2)}, N)$
${}_p\mathbf{X}_{(3)} \leftarrow$ `PartitionUnfoldedTensor`$(\mathbf{X}_{(3)}, N)$
${}_p\mathcal{X} \leftarrow$ `PartitionInputTensor`$(\mathcal{X}, N)$
for $t \leftarrow 1, \ldots, T$ **do**
  if $t = 1$ **then**
    initialize L sets of factor matrices $(\mathbf{A}_1, \mathbf{B}_1, \mathbf{C}_1), \ldots, (\mathbf{A}_L, \mathbf{B}_L, \mathbf{C}_L)$ randomly where $\mathbf{A}_i \in \mathbb{B}^{I \times R_1}$, $\mathbf{B}_i \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C}_i \in \mathbb{B}^{K \times R_3}$ for $i = 1, 2, \ldots, L$
    initialize L sets of core tensors $\mathcal{G}_1, \ldots, \mathcal{G}_L$ randomly
    $\mathcal{G}_i \leftarrow$ `UpdateCore`$({}_p\mathcal{X}, \mathcal{G}_i, \mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i)$ for $i = 1, 2, \ldots, L$
    apply `UpdateFactors` to each set $(\mathbf{A}_i, \mathbf{B}_i, \mathbf{C}_i, \mathcal{G}_i)$ for $i = 1, 2, \ldots, L$, and find the set s_{min} with the smallest error
    $(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G}) \leftarrow s_{min}$
  else
    $\mathcal{G} \leftarrow$ `UpdateCore`$({}_p\mathcal{X}, \mathcal{G}, \mathbf{A}, \mathbf{B}, \mathbf{C})$
    $(\mathbf{A}, \mathbf{B}, \mathbf{C}) \leftarrow$ `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G})$
  if *converged* **then**
    break out of **for** loop
return $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G}$
Function `UpdateFactors`$(\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathcal{G})$
  `/* minimize` $\left|\mathbf{X}_{(1)} - \mathbf{A} \boxtimes \mathbf{G}_{(1)} \boxtimes (\mathbf{C} \otimes \mathbf{B})^\top\right|$ `*/`
  $\mathbf{A} \leftarrow$ `UpdateFactorTucker`$({}_p\mathbf{X}_{(1)}, \mathbf{A}, \mathbf{C}, \mathbf{B}, \mathbf{G}_{(1)}, V)$
  `/* minimize` $\left|\mathbf{X}_{(2)} - \mathbf{B} \boxtimes \mathbf{G}_{(2)} \boxtimes (\mathbf{C} \otimes \mathbf{A})^\top\right|$ `*/`
  $\mathbf{B} \leftarrow$ `UpdateFactorTucker`$({}_p\mathbf{X}_{(2)}, \mathbf{B}, \mathbf{C}, \mathbf{A}, \mathbf{G}_{(2)}, V)$
  `/* minimize` $\left|\mathbf{X}_{(3)} - \mathbf{C} \boxtimes \mathbf{G}_{(3)} \boxtimes (\mathbf{B} \otimes \mathbf{A})^\top\right|$ `*/`
  $\mathbf{C} \leftarrow$ `UpdateFactorTucker`$({}_p\mathbf{X}_{(3)}, \mathbf{C}, \mathbf{B}, \mathbf{A}, \mathbf{G}_{(3)}, V)$
  return $\mathbf{A}, \mathbf{B}, \mathbf{C}$

entries in updating the core tensor. Partitioned unfolded tensor RDDs and the input tensor RDD 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 columnwise iteration, a driver node *collect*s errors computed from each partition to update columns. DBTF-TK updates the core tensor entry by entry. In each iteration,

Algorithm 9: `PartitionInputTensor`

Input: a three-way binary tensor $\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, and the number of partitions N.
Output: a partitioned input tensor ${}_p\mathcal{X} \in \mathbb{B}^{I \times J \times K}$.

distributed (D): split $\mathcal{X}$ into ${}_p\mathcal{X}$, which consists of non-overlapping subtensors ${}_p\mathcal{X}_1, {}_p\mathcal{X}_1, \ldots, {}_p\mathcal{X}_N$ where (1) ${}_p\mathcal{X}_t$ is associated with three ranges, I_t, J_t, and K_t, such that $|I_i| \times |J_i| \times |K_i| \approx |I_j| \times |J_j| \times |K_j|$ for all $i, j \in [1 \,..\, N]$; (2) ${}_p\mathcal{X}_t$ contains all $x_{ijk} \in \mathcal{X}$ for $i \in I_t$, $j \in J_t$, and $k \in K_t$; and (3) $\bigcup_{t=1}^{N} {}_p\mathcal{X}_t = \mathcal{X}$ and ${}_p\mathcal{X}_i \cap {}_p\mathcal{X}_j = \emptyset$ for all $i, j \in [1 \,..\, N]$ $(i \neq j)$
D: cache ${}_p\mathcal{X}$ across machines
return ${}_p\mathcal{X}$

Algorithm 10: `UpdateFactorTucker`

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), $\mathbf{M}_f \in \mathbb{B}^{Q \times R_f}$ (first matrix for the Kronecker product), and $\mathbf{M}_s \in \mathbb{B}^{S \times R_s}$ (second matrix for the Kronecker product), an unfolded core tensor $\mathbf{G}$, a threshold value V to limit the size of a single cache table, and the maximum percentage Z of zeros in the column being updated.
Output: an updated factor matrix $\mathbf{A}$.

`AugmentPartitionWithRowSummations`$({}_p\mathbf{X}, \mathbf{M}_s, V)$
$\mathbf{T}_g \leftarrow$ `GenerateRowSummations`$(\mathbf{G}, V)$
`/* iterate over columns and rows of` $\mathbf{A}$ `*/`
for *column iter* $c \leftarrow 1 \ldots R$ **do**
  for *row* $r \leftarrow 1 \ldots P$ **do**
    for $a_{rc} \leftarrow 0, 1$ **do**
      distributed: foreach *partition* $p' \in {}_p\mathbf{X}$ **do**
        $\mathbf{g} \leftarrow$ using $\mathbf{a}_{r:}$, fetch the cached Boolean row summation from $\mathbf{T}_g$ that corresponds to $\mathbf{a}_{r:} \boxtimes \mathbf{G}$
        foreach *block* $b \in p'$ **do**
          compute the cache key $\mathbf{k} \leftarrow \bigvee_{m=1}^{R_f} [\mathbf{M}_f]_{im} \wedge \mathbf{g}_{((m-1)R_s+1:mR_s)}$ where i is the row index of $\mathbf{M}_f$ such that block b is within the vertical boundaries of underlying $([\mathbf{M}_f]_{i:} \otimes \mathbf{M}_s)^\top$
          $\mathbf{v} \leftarrow$ using $\mathbf{k}$, fetch the cached Boolean row summation that corresponds to $[\mathbf{a}_{r:} \boxtimes \mathbf{G}] \boxtimes ([\mathbf{M}_f]_{i:} \otimes \mathbf{M}_s)^\top$
          compute the error between the fetched row $\mathbf{v}$ and the corresponding part of ${}_p\mathbf{x}_{r:}$
  collect errors for the entries of columns $\mathbf{a}_{:c}$ from all blocks (for both cases of when each entry is set to 0 and 1)
  for *row* $r \leftarrow 1 \ldots P$ **do** `/* update` $\mathbf{a}_{:c}$ `*/`
    update a_{rc} to the value that yields a smaller error (i.e., $\left|\mathbf{x}_{r:} - \mathbf{a}_{r:} \boxtimes \mathbf{G} \boxtimes (\mathbf{M}_f \otimes \mathbf{M}_s)^\top\right|$)
  if *the percentage of zeros in* $\mathbf{a}_{:c} > Z$ **then**
    find new values for a subset of rows which will make $\mathbf{a}_{:c}$ to obey Z with the smallest increase in error, and update those rows accordingly.
return A

Algorithm 11: `UpdateCore`

Input: a partitioned input tensor ${}_p\mathcal{X} \in \mathbb{B}^{I \times J \times K}$, a core tensor $\mathcal{G} \in \mathbb{B}^{R_1 \times R_2 \times R_3}$, and factor matrices $\mathbf{A} \in \mathbb{B}^{I \times R_1}$, $\mathbf{B} \in \mathbb{B}^{J \times R_2}$, and $\mathbf{C} \in \mathbb{B}^{K \times R_3}$.
Output: an updated core tensor $\mathcal{G}$.

for $(r_1, r_2, r_3) \in [1 \,..\, R_1] \times [1 \,..\, R_2] \times [1 \,..\, R_3]$ **do**
  $gain \leftarrow 0$
  $s_I, s_J, s_K \leftarrow$ rowwise sum of entries in $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ that are in those columns selected by entries in $\mathcal{G}$
  distributed: foreach *subtensor* ${}_p\mathcal{X}_t \in {}_p\mathcal{X}$ **do**
    $I_t, J_t, K_t \leftarrow$ three ranges associated with ${}_p\mathcal{X}_t$
    if $g_{r_1 r_2 r_3} = 0$ **then**
      foreach $i \in I_t$ *such that* $a_{ir_1} = 1$ **do**
        $c_I \leftarrow s_I(i) = 0$
        foreach $j \in J_t$ *such that* $b_{jr_2} = 1$ **do**
          $c_{IJ} \leftarrow c_I$ or $s_J(j) = 0$
          foreach $k \in K_t$ *such that* $c_{kr_3} = 1$ **do**
            $c_{IJK} \leftarrow c_{IJ}$ or $s_K(k) = 0$
            if c_{IJK} *and* ${}_p x_{ijk} = 1$ **then**
              $gain \leftarrow gain + 1$
              break out of all **foreach** loops
    else
      foreach $i \in I_t$ *such that* $a_{ir_1} = 1$ **do**
        if $s_I(i) \neq 1$ **then** continue
        foreach $j \in J_t$ *such that* $b_{jr_2} = 1$ **do**
          if $s_J(j) \neq 1$ **then** continue
          foreach $k \in K_t$ *such that* $c_{kr_3} = 1$ **do**
            if $s_K(k) \neq 1$ **then** continue
            if ${}_p x_{ijk} = 0$ **then**
              $gain \leftarrow gain + 1$
              break out of all **foreach** loops
  if $gain > 0$ **then**
    $g_{r_1 r_2 r_3} \leftarrow 1 - g_{r_1 r_2 r_3}$
return $\mathcal{G}$

executors process the partitioned input tensor in parallel using *foreachPartition* operation, computes the partial gains of flipping the current core tensor entry, and aggregates them using an *accumulator*. In order to upper bound the maximum percentage of zeros in the columns of the factor matrix being updated, we use a priority queue in which an inverse of the error of each candidate value (binary values assigned to the entries in each row which belong to the column being updated) is used as a priority. While the percentage of zeros is greater than the threshold, an element with the highest priority is popped off the priority queue and replaces the corresponding, current value provided that it decreases the percentage of zeros.

4.8 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 or

the dimension of each mode of a core tensor), M (number of machines), T (number of maximum iterations), N (number of partitions), V (maximum number of rows for caching, and Z (maximum percentage of zeros in the columns being updated). For the sake of simplicity, we assume an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$ and a core tensor $\mathcal{G} \in \mathbb{B}^{R \times R \times R}$ and that DBTF initializes a single set of factor matrices and a core tensor. Also, based on symbol definitions, we make simplifying assumptions that $N \ll I$, $R \ll I$, and $R^2 \leq I$. All proofs of the following lemmas appear in "Appendix."

4.8.1 Analysis of DBTF-CP

Lemma 4 *The time complexity of DBTF-CP is* $O\left(TI^3R\left\lceil\frac{R}{V}\right\rceil + TN\left\lceil\frac{R}{V}\right\rceil 2^{\lceil R/\lceil R/V\rceil\rceil}I\right)$.

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

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

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

4.8.2 Analysis of DBTF-TK

Lemma 8 *The time complexity of DBTF-TK is* $O\left(TI^3R^3 + TN\left\lceil\frac{R}{V}\right\rceil 2^{\lceil R/\lceil R/V\rceil\rceil}I\right)$.

Lemma 9 *The memory requirement of DBTF-TK is* $O(|\mathcal{X}| + (N+M)I\left\lceil\frac{R}{V}\right\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$.

Lemma 10 *The amount of shuffled data for partitioning an input tensor* $\mathcal{X}$ *is* $O(|\mathcal{X}|)$.

Lemma 11 *The amount of shuffled data after the partitioning of an input tensor* $\mathcal{X}$ *is* $O(TRI(M+N) + TR^3(MI+N) + TMR^2\left\lceil\frac{R}{V}\right\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$.

5 Experiments

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

- **Q1 Data Scalability (Section 5.2)** How well do DBTF and other methods scale up with respect to the following aspects of an input tensor: number of nonzeros, dimensionality, density, and rank?
- **Q2 Machine Scalability (Section 5.3)** How well does DBTF scale up with respect to the number of machines?
- **Q3 Reconstruction Error (Section 5.4)** How accurately do DBTF and other methods factorize the given tensor?

Table 3 Summary of real-world and synthetic tensors used for experiments

Name	I	J	K	Nonzeros
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 CP error	100	100	100	6.5K–240K
Synthetic TK error	100	100	100	1.5K–45K

B: billion, M: million, K: thousand

We introduce the datasets, baselines, and experimental environment in Sect. 5.1. After that, we answer the above questions in Sects. 5.2–5.4.

5.1 Experimental settings

5.1.1 Datasets

We use both real-world and synthetic tensors to evaluate the proposed method. The tensors used in experiments are listed in Table 3. 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'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.

[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.

5.1.2 Baselines

In experiments for Boolean CP factorization, we compare DBTF-CP with Walk'n'Merge [2] and BCP_ALS [3]. We also implemented an algorithm for Boolean CP decomposition of three-way binary data presented in [6], but found its results to be much worse than Walk'n'Merge and BCP_ALS (e.g., it takes three orders of magnitude more time than BCP_ALS and Walk'n'Merge for a tensor of size $I = J = K = 2^7$). So we omit reporting its results for the sake of clarity. In experiments for Boolean Tucker factorization, we compare DBTF-TK with Walk'n'Merge [2] and BTucker_ALS [3].

5.1.3 Environment

DBTF is implemented on the Apache Spark framework. We run experiments on a cluster with 17 machines, each of which is equipped with an Intel Xeon E3-1240v5 CPU (quad-core with hyperthreading at 3.50 GHz) and 32 GB RAM. The cluster runs Apache Spark v2.2.0 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. DBTF parameters L, V, and Z are set to 1, 15, and 0.95, respectively, and T is set to 10 for scalability tests and 20 for reconstruction error tests (see Algorithms 3, 5–8 and 10 for details on these parameters). We run Walk'n'Merge, BCP_ALS, and BTucker_ALS on one machine in the cluster. For the CP factorization by Walk'n'Merge, we use the original implementation[4] provided by the authors. However, the open-source implementation of Walk'n'Merge does not contain code for the Tucker factorization, which is obtained based on the CP factorization output of Walk'n'Merge. We implement the missing part, which is to merge the factor matrices returned from Walk'n'Merge and adjust the core tensor accordingly. We run Walk'n'Merge with the same parameter settings as described in [2]. The minimum size of blocks is set to 4-by-4-by-4. The length of random walks is set to 5. In reconstruction error tests, the merging threshold t is set to $1 - (n_d + 0.05)$ for CP factorization, and it is set to $1 - (n_d + 0.5)$ for Tucker factorization where n_d is the destructive noise level of an input tensor. Since Walk'n'Merge did not find blocks when it performs Tucker factorization with the threshold value used for CP factorization, we used smaller values for Tucker factorization. For scalability tests, t is set to 0.2 for both types of factorizations. Other parameters are set to the default values. We implement BCP_ALS and BTucker_ALS using the open-source code of ASSO[5] [48]. For ASSO, the threshold value for discretization is set to 0.7; default values are used for other parameters.

5.2 Data scalability

We evaluate the data scalability of DBTF and other methods for Boolean CP and Tucker factorizations using both synthetic random tensors (Sects. 5.2.1 and 5.2.2) and real-world tensors (Sects. 5.2.3 and 5.2.4).

5.2.1 Boolean CP factorization on synthetic data

We evaluate the data scalability of DBTF-CP, Walk'n'Merge, and BCP_ALS on synthetic tensors with respect to the dimensionality, density, and rank of a tensor. Experiments are allowed to run for up to 6 hours, and those running longer than that are marked as O.O.T. (Out Of Time).

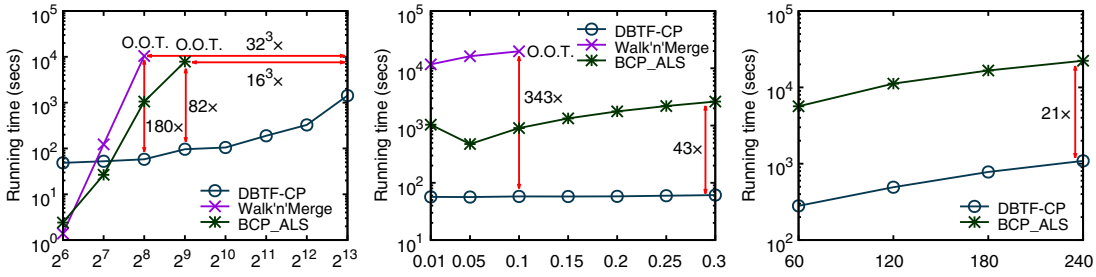
Dimensionality We increase the dimensionality $I = J = K$ of each mode from 2^6 to 2^{13}, while setting the tensor density to 0.01 and the rank R to 10. As shown in Fig. 7a, DBTF-CP successfully decomposes tensors of size $I = J = K = 2^{13}$, while Walk'n'Merge and BCP_ALS run out of time when $I = J = K \geq 2^9$ and $\geq 2^{10}$, respectively. Notice that the running time of Walk'n'Merge and BCP_ALS increases rapidly with the dimensionality: DBTF-CP decomposes the largest tensors Walk'n'Merge and BCP_ALS can process 180 × and 82 × faster than each method. DBTF-CP is slower than other methods for small tensors of 2^6 and 2^7 scale, 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 in these cases.

Density We increase the tensor density from 0.01 to 0.3, while fixing $I = J = K$ to 2^8 and the rank R to 10. As shown in Fig. 7b, DBTF-CP decomposes tensors of all densities and exhibits near constant performance regardless of the density. 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-CP runs 343× faster than Walk'n'Merge, and 43× faster than BCP_ALS. Also, the performance gap between DBTF-CP and BCP_ALS grows wider for tensors with greater density.

Rank We increase the rank R of a tensor from 60 to 240, while fixing $I = J = K$ to 2^8 and the tensor density to 0.01. V is set to 15 in all experiments. Walk'n'Merge is excluded from this experiment since its running time is constant across different ranks. As shown in Fig. 7c, both methods scale up to rank 240, and the running time increases almost linearly as the rank increases. When the rank is 240, DBTF-CP is 21× faster than BCP_ALS.

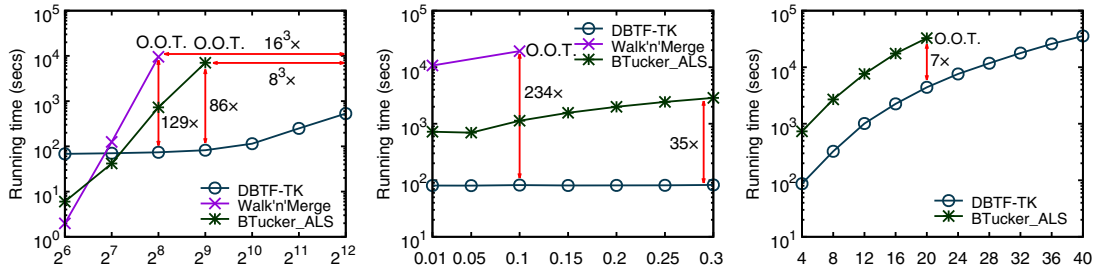
[4] http://people.mpi-inf.mpg.de/~pmiettin/src/walknmerge.zip.

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



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

Fig. 7 The scalability of DBTF-CP and other methods with respect to the dimensionality and density of a tensor, and the rank of CP decomposition. O.O.T.: Out Of Time (takes more than 6 h). DBTF-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180× less time (Fig. 7a). Overall, DBTF-CP achieves 21–343× speed up and exhibits near-linear scalability with regard to all data aspects



(a) Dimensionality. **(b)** Density. **(c)** CoreSize $R_1 = R_2 = R_3$.

Fig. 8 The scalability of DBTF-TK and other methods with respect to the dimensionality and density of a tensor, and the core size of Tucker decomposition. O.O.T.: Out Of Time (takes more than 12 h). DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129× less time (Fig. 8a). Overall, DBTF-TK achieves 7–234× speed up and exhibits near-linear scalability with regard to all data aspects

5.2.2 Boolean Tucker factorization on synthetic data

We evaluate the data scalability of DBTF-TK, Walk'n'Merge, and BTucker_ALS. Experiments that run longer than 12 hours are marked as O.O.T. (Out Of Time).

Dimensionality We increase the dimensionality $I = J = K$ of each mode from 2^6 to 2^{12} while setting the tensor density to 0.01 and the core size $R_1 = R_2 = R_3 = 4$ (Fig. 8a). While DBTF-TK is slower than BTucker_ALS and Walk'n'Merge for small tensors of 2^6 and 2^7 scale due to the overhead associated with a distributed system, the running time of BTucker_ALS and Walk'n'Merge increases much more rapidly than that of DBTF-TK. As a result, DBTF-TK is the only method that successfully decomposes tensors of $I = J = K = 2^{12}$, while Walk'n'Merge and BTucker_ALS run out of time when $I = J = K \geq 2^9$ and $\geq 2^{10}$, respectively. Furthermore, DBTF-TK decomposes the largest tensors that Walk'n'Merge and BTucker_ALS can handle 129× and 86× faster than each method.

Density We increase the tensor density from 0.01 to 0.3, while fixing $I = J = K$ to 2^8 and the core size $R_1 = R_2 = R_3$ to 4. Figure 8b shows that DBTF-TK decomposes tensors of all densities, and its running time remains almost the same as the density increases. BTucker_ALS also scales up to the tensor with 0.3 density. However, Walk'n'Merge runs out of time when the density becomes greater than 0.1, and even when it is 0.05. In terms of running time, DBTF-TK runs 234× and 35× faster than Walk'n'Merge and BTucker_ALS, respectively.

Rank We increase the core size $R_1 = R_2 = R_3$ from 4 to 40, while fixing $I = J = K$ to 2^8 and the tensor density to 0.01. V is set to 15 in all experiments. As shown in Fig. 8c, DBTF-TK scales up to the largest core size, while BTucker_ALS fails to scale up to core size greater than 20. Note that Walk'n'Merge is not shown in the figure since it does not allow users to specify the core size; instead, it automatically determines the core size according to the MDL principle. In terms of running time, DBTF-TK is faster than BTucker_ALS for all core sizes, with DBTF-TK being 7× faster than BTucker_ALS when core size $R_1 = R_2 = R_3$ is 20.

While DBTF-TK outperforms all baselines, the largest core size $R_1 = R_2 = R_3 = 40$ for Tucker factorization is much smaller than the largest rank size $R = 240$ used for CP factorization. This is because Tucker factorization is much more expensive than CP factorization as it involves all steps of CP factorization, and also performs steps to update a core

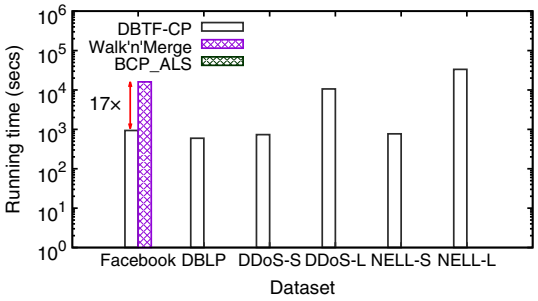


Fig. 9 The scalability of DBTF-CP and other methods on the real-world datasets. Notice that only DBTF-CP scales up to all datasets, while Walk'n'Merge processes only Facebook, and BCP_ALS fails to process all datasets. DBTF-CP runs 17× faster than Walk'n'Merge on Facebook. An empty bar denotes that the corresponding method runs out of time (> 12 h) or memory while decomposing the dataset

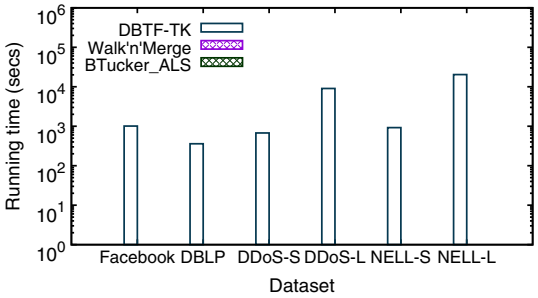


Fig. 10 The scalability of DBTF-TK and other methods on the real-world datasets. Notice that only DBTF-TK scales up to all datasets, while Walk'n'Merge and BTucker_ALS fail to process all datasets. An empty bar denotes that the corresponding method runs out of time (> 12 h) or memory while decomposing the dataset

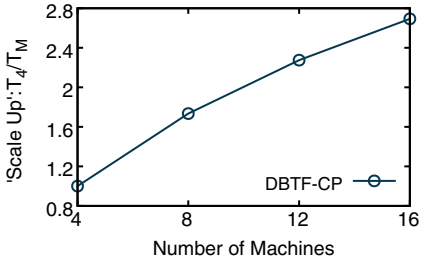
tensor, which is the most costly operation that takes time proportional to the cube of core size $R_1 = R_2 = R_3$ (see Lemma 8).

5.2.3 Boolean CP factorization on real-world data

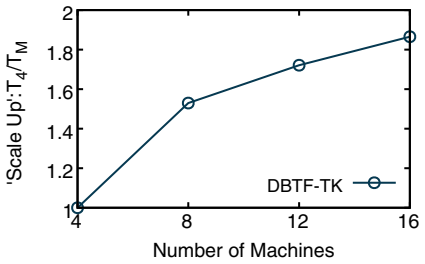
Figure 9 shows the running time of DBTF-CP, Walk'n'Merge, and BCP_ALS on real-world datasets. We set the maximum running time to 12 hours, and R to 10 for DBTF-CP and BCP_ALS. Among three methods, DBTF-CP is the only one that scales up for all datasets. Walk'n'Merge decomposes only Facebook and runs out of time for all other datasets; BCP_ALS fails to handle real-world tensors as it causes out-of-memory errors for all datasets, except for DBLP for which BCP_ALS runs out of time. Also, DBTF-CP runs 17× faster than Walk'n'Merge on Facebook.

5.2.4 Boolean Tucker factorization on real-world data

Figure 10 reports the running time of DBTF-TK, Walk'n'Merge, and BTucker_ALS on real-world tensors. We run each experiment for at most 12 hours with $R_1 = R_2 = R_3 = 4$. In Fig. 10, only DBTF-TK is shown as it is the only method that scales up to all datasets. While Walk'n'Merge finds blocks within the time limit for Facebook data, it runs out of time while merging factors and adjusting the core tensor. BTucker_ALS runs out of time for DBLP and causes out-of-memory errors for all other datasets.



(a) CP Factorization.



(b) Tucker Factorization.

Fig. 11 The scalability of DBTF-CP and DBTF-TK 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

5.3 Machine scalability

We measure the machine scalability of DBTF-CP and DBTF-TK 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.

Boolean CP factorization We use the synthetic tensor of size $I = J = K = 2^{12}$ and of density 0.01; we set the rank R to 10. Figure 11a shows that DBTF-CP scales up near linearly. Overall, DBTF-CP achieves 2.69 × speed up, as the number of machines increases fourfold.

Boolean Tucker factorization We use the synthetic tensor of size $I = J = K = 2^{11}$ and of density 0.01; we set the core size $R_1 = R_2 = R_3$ to 4. Figure 11b shows that DBTF-TK shows near-linear scalability, achieving 1.87 × speed up when the number of machines is increased from 4 to 16.

5.4 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 input tensor and $\mathcal{X}'$ is a reconstructed tensor. Tensors of size

$I = J = K = 100$ are used in experiments. We run each configuration three times and report the average of the results. We compare DBTF with Walk'n'Merge as they take different approaches for Boolean CP and Tucker decompositions and exclude BCP_ALS and BTucker_ALS as DBTF, BCP_ALS, and BTucker_ALS are based on the same Boolean decomposition frameworks.

5.4.1 Boolean CP factorization

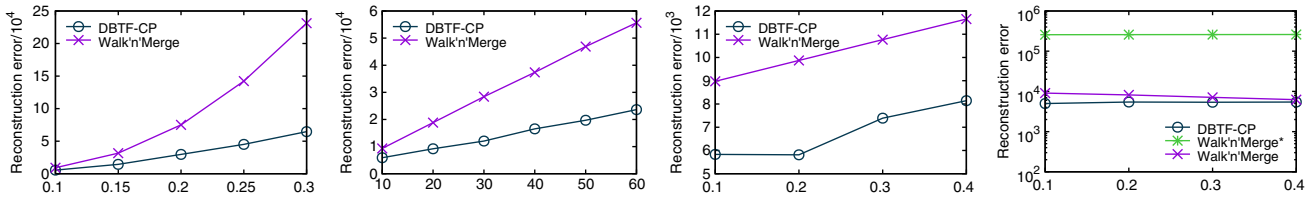
We measure reconstruction errors, varying one of the four different data aspects—factor matrix density (0.1), rank (10), additive noise level (0.1), and destructive noise level (0.1)—while fixing the others to the default values. The values in the parentheses are the default settings for each aspect. For Walk'n'Merge, we report the reconstruction error computed from the blocks obtained before the second part of the merging phase [2], since the subsequent merging procedure significantly increased the reconstruction error when applied to our synthetic tensors. Figure 12d shows the difference between the version of Walk'n'Merge with the second part of 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 Fig. 12a, the reconstruction error of DBTF-CP is smaller than that of Walk'n'Merge for all densities. In particular, as the density increases, the gap between DBTF-CP and Walk'n'Merge widens.

Rank We increase the rank of a tensor from 10 to 60. As shown in Fig. 12b, 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 nonzeros in the input tensor. Notice that the reconstruction error of DBTF-CP is smaller than that of Walk'n'Merge for all ranks.

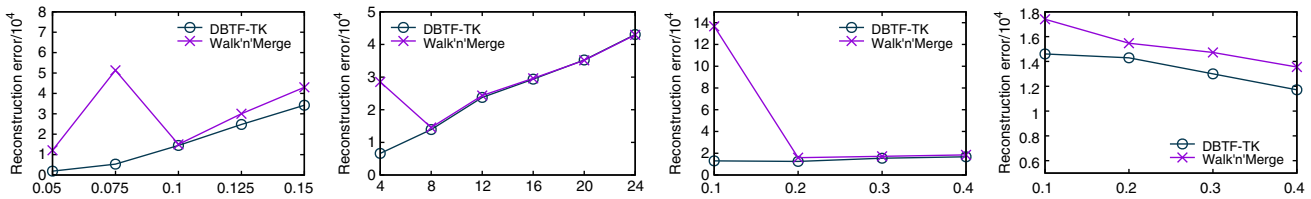
Additive noise level We increase the additive noise level from 0.1 to 0.4. As shown in Fig. 12c, the reconstruction errors of both methods increase in proportion to the additive noise level. While the relative accuracy improvement obtained with DBTF-CP tends to decrease as the noise level increases, the reconstruction error of DBTF-CP is smaller than that of Walk'n'Merge for all additive noise levels.

Destructive noise level We increase the destructive noise level from 0.1 to 0.4. Figure 12d shows that DBTF-CP produces more accurate results than Walk'n'Merge across all destructive noise levels. As the destructive noise level increases, the reconstruction error of DBTF-CP slightly increases, while that of Walk'n'Merge decreases; as a result, the gap between two methods becomes smaller. Destructive noise makes the factorization harder by sparsifying tensors and introducing noises at the same time.



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

Fig. 12 The reconstruction error of DBTF-CP 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-CP are smaller than those of Walk'n'Merge for all aspects



(a) Factor Matrix Density. **(b)** Core Size $R_1 = R_2 = R_3$. **(c)** Additive Noise Level. **(d)** Destructive Noise Level.

Fig. 13 The reconstruction error of DBTF-TK and other methods with respect to factor matrix density, core size, additive noise level, and destructive noise level. Across all aspects, the reconstruction errors of DBTF-TK are smaller than or close to those of Walk'n'Merge. Note that, in contrast to DBTF-TK, Walk'n'Merge shows performance fluctuations in a few cases

5.4.2 Boolean Tucker factorization

We measure reconstruction errors, varying one of the following data aspects—factor matrix density (0.1), core size ($R_1 = R_2 = R_3 = 8$), additive noise level (0.2), and destructive noise level (0.2)—while fixing the others to their default values. The values in the parentheses are the default settings for each aspect.

Factor matrix density We increase the density of factor matrices from 0.05 to 0.15. As shown in Fig. 13a, the reconstruction error of DBTF-TK is smaller than or close to that of Walk'n'Merge across all densities. Note that, in contrast to DBTF-TK, Walk'n'Merge shows performance fluctuation when the density is 0.075.

Rank We increase the core size $R_1 = R_2 = R_3$ from 4 to 24. As shown in Fig. 13b, the reconstruction errors of both methods increase in proportion to the core size, and DBTF-TK and Walk'n'Merge exhibit similar performance.

Additive noise level We increase the additive noise level from 0.1 to 0.4. Figure 13c shows that both methods perform similarly as the noise level increases, except when the additive noise level is 0.1, in which case the reconstruction error of Walk'n'Merge is approximately 10× greater than that of DBTF-TK.

Destructive noise level We increase the destructive noise level from 0.1 to 0.4. Figure 13d shows that the reconstruction errors of both methods decrease as the noise level is increased, and DBTF-TK is consistently more accurate than Walk'n'Merge for all destructive noise levels.

6 Conclusion

In this paper, we propose DBTF, a distributed method for Boolean CP (DBTF-CP) and Tucker (DBTF-TK) factorizations running on the Apache Spark framework. By distributed data generation with minimal network transfer, 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-CP decomposes up to 16^3–$32^3\times$ larger tensors than existing methods in 82–180× less time, and DBTF-TK decomposes up to 8^3–$16^3\times$ larger tensors than existing methods in 86–129× less time. Furthermore, both DBTF-CP and DBTF-TK exhibit near-linear scalability in terms of tensor dimensionality, density, rank, and the number of machines.

Acknowledgements 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). The ICT at Seoul National University provides research facilities for this study. The Institute of Engineering Research at Seoul National University provided research facilities for this work.

A Proofs

A.1 Proof of Lemma 4

Proof Algorithm 3 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-CP splits $\mathbf{X}$ into N partitions, and further divides each partition into a set of blocks (Algorithm 4). Unfolding takes $O(|\mathcal{X}|)$ time as each entry can be mapped in constant time (Eq. (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) Random initialization of factor matrices takes $O(IR)$ time.
(3) The update of a factor matrix (Algorithm 5) consists of the following steps (i, ii, iii, and iv):

i. Caching row summations of a factor matrix (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 cached row summations (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, the time complexity for this step is $O(IR \lceil \lceil \frac{R}{V} \rceil \min(V, R) \max(I, N) + (\lceil \frac{R}{V} \rceil - 1)I^2 \rceil)$. Simplifying terms, we get $O(I^3 R \lceil \frac{R}{V} \rceil)$.
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-CP

constructs row summations ($\mathbf{a}_{r:} \boxtimes (\mathbf{M}_f \odot \mathbf{M}_s)^\top$ in Algorithm 5) twice (for $a_{rc} = 0$ and 1). Therefore, given a rank R, this step takes $O(I^3 R)$ time.

iv. Updating a factor matrix (lines 10–14). Updating an entry in a factor matrix requires summing up errors for each value collected from N partitions, which takes $O(N)$ time. Updating all entries takes $O(NIR)$ time. In case the percentage of zeros in the column being updated is greater than Z, an additional step is performed to make the sparsity of the column less than Z, which takes $O(I \log(I))$ time as all $2I$ values may need to be fetched in the order of increasing error in the worst case. Since we have R columns, this additional step takes $O(RI \log(I))$ in total. Thus, step iv takes $O(NIR + RI \log(I))$ time.

After simplifying terms, DBTF-CP's time complexity is $O\left(TI^3R\left\lceil\frac{R}{V}\right\rceil + TN\left\lceil\frac{R}{V}\right\rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I\right)$. At each iteration, the dominating term is $O(I^3 R)$ that comes from fetching row summation and calculating its error (steps ii and iii), which is an $O(I^2)$ operation that is performed $2IR$ times. Note that the worst-case time complexity for this error calculation is $O(I^2)$ even when the input tensor is sparse because the time for this operation depends not only on the nonzeros in the row of an input tensor, but also on the nonzeros in the corresponding row of the intermediate matrix product (e.g., $(\mathbf{C} \odot \mathbf{B})^\top$), which could be full of nonzeros in the worst case. However, given sparse tensors in practice, factor matrices are updated to be sparse such that the reconstructed tensor gets closer to the sparse input tensor, which makes the time required for the dominating operation much less than $O(I^2)$.

A.2 Proof of Lemma 5

Proof For the decomposition of an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$, DBTF-CP stores the following four types of data in memory at each iteration: (1) partitioned unfolded input 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-CP 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 row summations of a factor matrix 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 two errors for the entries of the column being updated, which takes $O(NI)$ memory.

A.3 Proof of Lemma 6

Proof DBTF-CP 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 $O(|\mathcal{X}|)$.

A.4 Proof of Lemma 7

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-CP 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-CP 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 $O(NIR)$ space in total. Accordingly, the total amount of data shuffled for T iterations after partitioning $\mathcal{X}$ is $O(TRI(M + N))$.

A.5 Proof of Lemma 8

Proof Algorithm 8 is composed of four operations: (1) partitioning (lines 1–4), (2) initialization (lines 7–8), (3) updating factor matrices (lines 10 and 14), and (4) updating a core tensor (lines 9 and 13).

(1) Partitioning of an input tensor $\mathcal{X}$ into ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ (lines 1–3) takes $O(|\mathcal{X}|)$ time as in DBTF-CP. Similarly, partitioning of $\mathcal{X}$ into ${}_p\mathcal{X}$ (line 4) takes $O(|\mathcal{X}|)$ time since determining which partition an entry of $\mathcal{X}$ belongs to can be done in constant time.
(2) Randomly initializing factor matrices and a core tensor takes $O(IR)$ and $O(R^3)$ time, respectively.
(3) The update of a factor matrix (Algorithm 10) consists of the following steps (i, ii, iii, and iv):

i. Caching row summations (lines 1–2). Caching row summations of a factor matrix takes $O(N\lceil\frac{R}{V}\rceil$

$2^{\lceil R/\lceil R/V\rceil\rceil} I$) as in DBTF-CP. Caching row summations of an unfolded core tensor requires $O(\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil} R^2)$ as a single row summation can be computed in $O(R^2)$ time. Assuming $R^2 \leq I$, this step requires $O(N \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I)$ time.

ii. Fetching cached row summations (lines 9–10). (a) Row summations of an unfolded core tensor are fetched $O(IR)$ times in each partition. If $R \leq V$, a row summation can be obtained with one access to the cache. If $R > V$, multiple accesses are required to fetch row summations from $\lceil \frac{R}{V} \rceil$ tables, and there is an additional cost to sum up $\lceil \frac{R}{V} \rceil$ row summations. In sum, this operation takes $O(NIR[\lceil \frac{R}{V} \rceil + (\lceil \frac{R}{V} \rceil - 1)R^2])$ time. (b) Fetching a cached row summation of a factor matrix is identical to that in DBTF-CP, except for the computation of cache key, which takes $O(R^2)$ time. Therefore, this operation takes $O(IR[\lceil \frac{R}{V} \rceil R^2 \max(I, N) + (\lceil \frac{R}{V} \rceil - 1)I^2])$ in total. Simplifying (a) and (b) under the assumption that $R^2 \leq I$ and $\max(I, N) = I$, the time complexity for this step reduces to $O(I^3 R \lceil \frac{R}{V} \rceil)$.

iii. Computing the error for the fetched row summation (line 11). This step takes the same time as in DBTF-CP, which is $O(I^3 R)$.

iv. Updating a factor matrix (lines 12–16). This step takes the same time as in DBTF-CP, which is $O(NIR + RI \log(I))$.

(4) For the update of a core tensor (Algorithm 11), two operations are repeatedly performed for each core tensor entry. First, rowwise sum of entries in factor matrices are computed (line 3), which takes $O(IR)$ time. Second, DBTF-TK determines whether flipping the core tensor entry would improve accuracy (lines 4–25). This step takes $O(I^3)$ time in the worst case when the factor matrices are full of nonzeros. In sum, it takes $O(I^3 R^3)$ to update a core tensor.

In sum, DBTF-TK takes $O\left(TI^3R^3 + TN \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil} I\right)$ time.

A.6 Proof of Lemma 9

Proof In order to decompose an input tensor $\mathcal{X} \in \mathbb{B}^{I \times I \times I}$, DBTF-TK stores the following five types of data in memory at each iteration: (1) partitioned input tensors ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, ${}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$, (2) row summation results, (3) a core tensor $\mathcal{G}$, (4) factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, and (5) errors for the entries of a column being updated.

(1) Since partitioning does not change the total number of elements, ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, ${}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$ require $O(|\mathcal{X}|)$ memory.

(2) Two types of row summation results are maintained in DBTF-TK: the first for the factor matrix (e.g., $\mathbf{B}^\top$), and the second for the unfolded core tensors (e.g., $\mathbf{G}_{(1)}$). Note that, given R number of rows, the total number of row summations to be cached is $O(\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ by Lemma 2. First, the cache tables for the factor matrix are the same as those used in DBTF-CP; thus, they use $O(NI \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ memory. Second, across M machines, the cache tables for the unfolded core tensor require $O(MR^2 \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ as a single entry uses $O(R^2)$ space. Assuming $R^2 \leq I$, $O((N + M)I \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$ memory is required in total for row summation results.

(3) Since the core tensor $\mathcal{G}$ is broadcast to each machine, $O(MR^3)$ is required.

(4) Factor matrices require $O(MRI)$ memory as in DBTF-CP.

(5) $O(NI)$ memory is required since each partition stores two errors for each entry of the column being updated as in DBTF-CP.

A.7 Proof of Lemma 10

Proof In DBTF-TK, an input tensor $\mathcal{X}$ is partitioned in four different ways, where the first three are ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, and ${}_p\mathbf{X}_{(3)}$ that are used for updating factor matrices, and the last one is ${}_p\mathcal{X}$ that is used for updating a core tensor. Each machine is assigned non-overlapping partitions of the input tensor. The entire data can be shuffled in the worst case, depending on the data distribution. Thus, the total amount of data shuffled for partitioning $\mathcal{X}$ is $O(|\mathcal{X}|)$.

A.8 Proof of Lemma 11

Proof As in DBTF-CP, partitioned input tensors ${}_p\mathbf{X}_{(1)}$, ${}_p\mathbf{X}_{(2)}$, ${}_p\mathbf{X}_{(3)}$, and ${}_p\mathcal{X}$ are shuffled only once in the beginning. After that, DBTF-TK performs data shuffling at each iteration in order to update (1) factor matrices $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, and (2) a core tensor $\mathcal{G}$.

(1) In updating factor matrices, DBTF-TK uses all data used in DBTF-CP, which is $O(TRI(M + N))$. Also, DBTF-TK broadcasts the tables containing the combinations of row summations of three unfolded core tensors ($\mathbf{G_{(1)}}$, $\mathbf{G_{(2)}}$, and $\mathbf{G_{(3)}}$) to each machine at every iteration. Since, given R, the total number of row summations to be cached is $O(\lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$, and each row summation uses $O(R^2)$ space, broadcasting these tables overall requires $O(TMR^2 \lceil \frac{R}{V} \rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$.

(2) DBTF-TK broadcasts the rowwise sum of entries in factor matrices to each machine when a core tensor $\mathcal{G}$ is updated, which takes $O(MIR^3)$ space in each iteration. Also, in updating an element of $\mathcal{G}$, DBTF-TK aggre-

gates partial gains computed from each partition, which requires $O(NR^3)$ for each iteration.

Accordingly, the total amount of data shuffled for T iterations after partitioning $\mathcal{X}$ is $O(TRI(M+N)+TR^3(MI+N)+TMR^2\lceil\frac{R}{V}\rceil 2^{\lceil R/\lceil R/V\rceil\rceil})$.

References

1. Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)
2. Erdős, D., Miettinen, P.: Walk 'n' merge: a scalable algorithm for Boolean tensor factorization. In: ICDM, pp. 1037–1042 (2013)
3. Miettinen, P.: Boolean tensor factorizations. In: ICDM (2011)
4. Erdős, D., Miettinen, P.: Discovering facts with Boolean tensor tucker decomposition. In: CIKM, pp. 1569–1572 (2013)
5. Metzler, S., Miettinen, P.: Clustering Boolean tensors. DMKD **29**(5), 1343–1373 (2015)
6. Belohlávek, R., Glodeanu, C.V., Vychodil, V.: Optimal factorization of three-way binary data using triadic concepts. Order **30**(2), 437–454 (2013)
7. Leenen, I., Van Mechelen, I., De Boeck, P., Rosenberg, S.: INDCLAS: a three-way hierarchical classes model. Psychometrika **64**(1), 9–24 (1999)
8. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI, pp. 15–28 (2012)
9. Park, N., Oh, S., Kang, U.: Fast and scalable distributed Boolean tensor factorization. In: ICDE, pp. 1071–1082 (2017)
10. Cerf, L., Besson, J., Robardet, C., Boulicaut, J.: Closed patterns meet n-ary relations. TKDD **3**(1), 3 (2009)
11. Ji, L., Tan, K., Tung, A.K.H.: Mining frequent closed cubes in 3D datasets. In: VLDB, pp. 811–822 (2006)
12. Kang, U., Papalexakis, E.E., Harpale, A., Faloutsos, C.: Gigatensor: scaling tensor analysis up by 100 times—algorithms and discoveries. In: KDD, pp. 316–324 (2012)
13. Jeon, B., Jeon, I., Sael, L., Kang, U.: Scout: scalable coupled matrix-tensor factorization—algorithm and discoveries. In: ICDE, pp. 811–822 (2016)
14. Jeon, I., Papalexakis, E.E., Faloutsos, C., Sael, L., Kang, U.: Mining billion-scale tensors: algorithms and discoveries. VLDB J. **25**(4), 519–544 (2016)
15. Sael, L., Jeon, I., Kang, U.: Scalable tensor mining. Big Data Res. **2**(2), 82–86 (2015). **(visions on Big Data)**
16. Park, N., Jeon, B., Lee, J., Kang, U.: Bigtensor: mining billion-scale tensor made easy. In: CIKM, pp. 2457–2460 (2016)
17. Beutel, A., Talukdar, P.P., Kumar, A., Faloutsos, C., Papalexakis, E.E., Xing, E.P.: Flexifact: scalable flexible factorization of coupled tensors on hadoop. In: SDM, pp. 109–117 (2014)
18. Papalexakis, E.E., Faloutsos, C., Sidiropoulos, N.D.: Parcube: sparse parallelizable tensor decompositions. In: ECML PKDD, pp. 521–536 (2012)
19. Li, J., Choi, J., Perros, I., Sun, J., Vuduc, R.: Model-driven sparse CP decomposition for higher-order tensors. In: IPDPS (2017)
20. Smith, S., Park, J., Karypis, G.: An exploration of optimization algorithms for high performance tensor completion. In: SC (2016)
21. Karlsson, L., Kressner, D., Uschmajew, A.: Parallel algorithms for tensor completion in the CP format. Parallel Comput. **57**, 222–234 (2016)
22. Shin, K., Sael, L., Kang, U.: Fully scalable methods for distributed tensor factorization. TKDE **29**(1), 100–113 (2017)
23. Lathauwer, L.D., Moor, B.D., Vandewalle, J.: On the best rank-1 and rank-(R_1, R_2, ..., R_N) approximation of higher-order tensors. SIMAX **21**(4), 1324–1342 (2000)
24. Kolda, T.G., Sun, J.: Scalable tensor decompositions for multi-aspect data mining. In: ICDM, pp. 363–372 (2008)
25. Oh, J., Shin, K., Papalexakis, E.E., Faloutsos, C., Yu, H.: S-hot: scalable high-order tucker decomposition. In: WSDM (2017)
26. Smith, S., Karypis, G.: Accelerating the tucker decomposition with compressed sparse tensors. In: Europar (2017)
27. Kaya, O., Uçar, B.: High performance parallel algorithms for the tucker decomposition of sparse tensors. In: ICPP (2016)
28. Oh, S., Park, N., Sael, L., Kang, U.: Scalable tucker factorization for sparse tensors—algorithms and discoveries. In: ICDE (2018)
29. Chakaravarthy, V.T., Choi, J.W., Joseph, D.J., Liu, X., Murali, P., Sabharwal, Y., Sreedhar, D.: On optimizing distributed tucker decomposition for dense tensors (2017). CoRR arXiv:1707.05594
30. Choi, J.H., Vishwanathan, S.: Dfacto: distributed factorization of tensors. In: NIPS (2014)
31. Kaya, O., Uçar, B.: Scalable sparse tensor decompositions in distributed memory systems. In: SC, pp. 1–11 (2015)
32. Austin, W., Ballard, G., Kolda, T.G.: Parallel tensor compression for large-scale scientific data. In: IPDPS (2016)
33. Smith, S., Karypis, G.: A medium-grained algorithm for distributed sparse tensor factorization. In: IPDPS (2016)
34. Acer, S., Torun, T., Aykanat, C.: Improving medium-grain partitioning for scalable sparse tensor decomposition. IEEE Trans. Parallel Distrib. Syst. **29**, 2814–2825 (2018)
35. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)
36. Apache hadoop. http://hadoop.apache.org/
37. Kang, U., Tsourakakis, C.E., Faloutsos, C.: PEGASUS: a peta-scale graph mining system. In: ICDM, pp. 229–238 (2009)
38. Park, H.-M., Park, N., Myaeng, S.-H., Kang, U.: Partition aware connected component computation in distributed systems. In: ICDM (2016)
39. Park, H.-M., Myaeng, S.-H., Kang, U.: Pte: enumerating trillion triangles on distributed systems. In: KDD, pp. 1115–1124 (2016)
40. Kang, U., Tong, H., Sun, J., Lin, C., Faloutsos, C.: GBASE: a scalable and general graph management system. In: KDD
41. Kalavri, V., Vlassov, V.: Mapreduce: limitations, optimizations and open issues. In: TrustCom, pp. 1031–1038 (2013)
42. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: HotCloud (2010)
43. Lulli, A., Ricci, L., Carlini, E., Dazzi, P., Lucchese, C.: Cracker: crumbling large graphs into connected components. In: ISCC, pp. 574–581 (2015)
44. Wiewiórka, M.S., Messina, A., Pacholewska, A., Maffioletti, S., Gawrysiak, P., Okoniewski, M.J.: Sparkseq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision. Bioinformatics **30**(18), 2652–2653 (2014)
45. Gu, R., Tang, Y., Wang, Z., Wang, S., Yin, X., Yuan, C., Huang, Y.: Efficient large scale distributed matrix computation with spark. In: IEEE BigData, pp. 2327–2336 (2015)
46. Zadeh, R.B., Meng, X., Ulanov, A., Yavuz, B., Pu, L., Venkataraman, S., Sparks, E.R., Staple, A., Zaharia, M.: Matrix computations and optimization in apache spark. In: KDD, pp. 31–38 (2016)
47. Kim, H., Park, J., Jang, J., Yoon, S.: Deepspark: spark-based deep learning supporting asynchronous updates and caffe compatibility (2016). CoRR arXiv:1602.08191
48. Miettinen, P., Mielikäinen, T., Gionis, A., Das, G., Mannila, H.: The discrete basis problem. TKDE **20**(10), 1348–1362 (2008)