DisMASTD: An Efficient Distributed Multi-Aspect Streaming Tensor Decomposition

Keyu Yang[†] Yunjun Gao[†#] Yifeng Shen[†] Baihua Zheng[‡] Lu Chen[⋆]
[†]*College of Computer Science, Zhejiang University, China*
[‡]*School of Information Systems, Singapore Management University, Singapore*
[⋆]*Department of Computer Science, Aalborg University, Denmark*
[#]*Alibaba-Zhejiang University Joint Institute of Frontier Technologies, China*
{*kyyang, gaoyj, yfshen*}*@zju.edu.cn* *bhzheng@smu.edu.sg* *luchen@cs.aau.dk*

***Abstract*—Tensor decomposition is a fundamental multi-dimensional data analysis tool for many data-driven applications, such as social computing, computer vision, and bioinformatics, to name but a few. However, the rapidly increasing streaming data nowadays introduces new challenges to traditional static tensor decomposition. It requires an efficient distributed dynamic tensor decomposition without re-computing the whole tensor from scratch. In this paper, we propose DisMASTD, an efficient distributed multi-aspect streaming tensor decomposition. First, we prove the optimal tensor partitioning problem is NP-hard. Second, we present two heuristic tensor partitioning approaches to ensure the load balancing. Third, we develop a distributed multi-aspect streaming tensor decomposition computation method, which avoids repetitive computation and reduces network communication by maintaining and reusing the intermediate results. Last but not least, we perform extensive experiments with both real and synthetic datasets to demonstrate the efficiency and scalability of DisMASTD.**

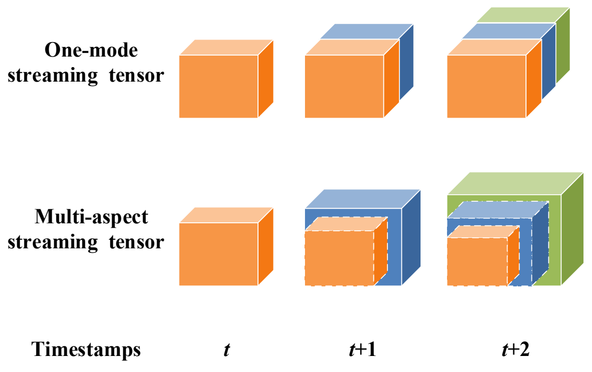


Fig. 1. Two categories of streaming tensors

I. Introduction

Multi-dimensional data rapidly and incrementally arises in many real-life applications. A natural representation of this kind of data is called *tensor*, which is an extension of a two-dimensional matrix to three or higher dimensions. Tensor decomposition, which aims at discovering the latent representations, has received much attention [1]–[4]. It is used in many data-driven applications, ranging from social computing (e.g., recommendation system [5], link prediction [6], and urban computing [7]) and computer vision (e.g., image/video completion [8] and compression [9]) to healthcare and medical applications [10]. Here, we present an example.

Recommendation System Application: Recommendation system aims at predicting the users' preferences based on users' past behaviors as well as decisions made by other similar users. If we use the quaternary tuple $\langle u, p, t, r\rangle$ to represent the fact that the user u gives the rating r to the product p at time t, we can model the prediction of users' preferences (i.e., ratings) on products as a tensor decomposition problem, i.e., treating the predicted "ratings" as missing entries of data tensors that could be complemented by the latent representations after tensor decomposition.

Beyond the traditional static setting, applications in the real world nowadays are producing high volume streaming data. According to *Data Never Sleeps Report*[1], over 2.5×10^{18} bytes of data are created every single day. In every minute, Snapchat users share 2,083,333 photos, Twitter users publish 473,400 tweets, and Skype users make 176,220 calls. These example digits showcase that not only the large volume but also the high velocity of data generate nowadays. Considering the continuous expansion nature of the data, it is definitely not feasible to perform tensor reconstruction from scratch whenever the data is updated [11]. The overwhelmingly increasing data motivates us to investigate an efficient distributed dynamic tensor decomposition method.

The existing efforts on tensor decomposition mostly focus on the static setting [12]–[16]. They all suffer from a common limitation, i.e., they are not able to tackle the dynamic tensor setting. To address this, dynamic tensor decomposition methods have been proposed. However, they mainly aim at a widely used assumption that tensors will be developed in only one dimension (or one *mode*). Following this oversimplified assumption, online methods have been presented in [17], [18].

Nevertheless, in many real-life applications, a tensor could be developed in *multiple modes*. Recall the example about recommendation system, both users and products can increase over time. To this end, the problem of *multi-aspect streaming tensor* has been studied in [19], [20]. Fig. 1 illustrates the

[1]https://www.domo.com/solution/data-never-sleeps-6

differences between the traditional one-mode streaming tensor and the multi-aspect streaming tensor. From timestamps t to $t+2$, the multi-aspect streaming tensor develops in all three modes, while the tensor in the traditional streaming setting only develops in one mode. Existing studies on multi-aspect streaming tensor aim at a *centralized environment*. They have limited scalability to support large data sets. Motivated by the limitations of existing methods, we dedicate this paper to the development of *Distributed Multi-Aspect Streaming Tensor Decomposition* (*DisMASTD* for short). DisMASTD is, to the best of our knowledge, the first attempt to perform the decomposition of multi-aspect streaming tensors in a *distributed environment* that aims to improve the scalability. Towards this, there are two challenges to be addressed.

First, *how to ensure load balancing among all the worker nodes in a distributed environment?* Load balancing is a basic requirement in distributed systems. We formulate an optimal tensor partitioning problem as the solution. Unfortunately, this problem is *NP-hard*. Therefore, we utilize two heuristic tensor partitioning methods, i.e., the *Greedy Tensor Partitioning algorithm* (*GTP*) and the *Max-min fit Tensor Partitioning method* (*MTP*). GTP predetermines the number of non-zero elements, namely, *target*, in each tensor partition, and derives the tensor partitioning to reach the target. MTP aims to split the tensor such that each tensor partition has roughly-equal number of non-zero elements.

Second, *how to reduce the computational cost during tensor decomposition?* The bottleneck cost of tensor decomposition is an operator, called *Marticized Tensor Times Khatri-Rao Product* (MTTKRP). DisMASTD avoids repetitive computation, and reduces network communication by maintaining and reusing MTTKRP results.

To sum up, the key contributions of this paper are summarized as follows:

- We propose DisMASTD, an efficient distributed multi-aspect streaming tensor decomposition. To our knowledge, it is the first attempt to tackle this problem.
- We prove the optimal partitioning problem is NP-hard and utilize two heuristic tensor partitioning methods, i.e., GTP and MTP, to ensure load balancing in a distributed environment.
- We design an efficient computation method for the distributed multi-aspect streaming tensor decomposition, which is able to avoid repetitive computation and reduce the network communication.
- We conduct extensive experiments using both real and synthetic datasets to verify the efficiency and scalability of DisMASTD.

The rest of the paper is organized in the following. Section II reviews related work about tensor decomposition. Section III introduces the definitions related to multi-aspect streaming tensor decomposition. Section IV elaborates DisMASTD, the solution to distributed multi-aspect streaming tensor decomposition studied in this work. Section V reports experimental results and our findings. Finally, Section VI concludes the paper.

II. Related Work

In this section, we overview previous studies on tensor decomposition under both static and dynamic settings.

A. Tensor Decomposition in Static Setting

CANDECOMP/PARAFAC (CP) and Tucker decompositions are two main widely used tensor decomposition algorithms. CP decomposition is proposed by Hitchcock [21] and then further enhanced by Carroll and Chang [22] as well as Harshman [23]. CP decomposition of an N^{th}-order tensor is an approximation of outer products for N loading matrices. Tucker decomposition is originally proposed by Tucker [24] and then is further developed by Kroonenberg and Leeuw [25] as well as Lathauwer et al. [26]. Unlike CP, Tucker decomposes an N^{th}-order tensor into N factor matrices, which are multiplied by a core tensor. There are three most commonly used optimizing algorithms for tensor decomposition, including Alternating Least Square (ALS) [27], [28], Stochastic Gradient Descent (SGD) [29], [30], and Coordinate Descent (CDD) [31], [32].

Distributed static tensor decomposition has been studied in the era of big data. GigaTensor [13] is a large-scale tensor decomposition algorithm on MapReduce. Scout [33] considers scalable coupled matrix-tensor factorization on MapReduce for analyzing large tensors with additional information. The above two tensor decomposition methods have been integrated into BIGtensor [34], a unified library for tensor data mining. SPLATT [16], [35] is another library for parallel tensor factorization. CDTF and SALS [15] are two tensor factorization algorithms for high-order and large-scale tensors in distributed environments. DisTenC [12] is a distributed tensor completion algorithm with auxiliary information on Spark. CartHP [36] improves scalable sparse tensor decomposition by hypergraph-based partitioning. Moreover, there are several existing studies that aim at tensor factorization with specific characteristics, such as sparse [14] and boolean tensor [37], [38].

Nevertheless, all the aforementioned algorithms are designed under the traditional static tensor setting, and thus, they cannot efficiently handle tensor decomposition for newly emerging data.

B. Tensor Decomposition in Dynamic Setting

The increasing amount of dynamic data nowadays motivates the studies of dynamic tensor decomposition. Existing efforts on dynamic tensor decomposition mostly focus on traditional streaming tensors, which are assumed to be developed in only one mode.

As for CP decomposition, Nion and Sidiropoulos [39] propose two adaptive PARAFAC algorithms. One, termed as PARAFAC-SDT, is based on simultaneous diagonalization. The other, called PARAFAC-RLST, utilizes weighted least squares to track the online third-order tensor decomposition. Phan and Cichocki [40] divide large-scale tensors into grids, and propose the tensor factorization method, which could be used in dynamic tensor factorization. Mardani et al. [17] leverage rank minimization and subspace learning to enable

TABLE I
COMPARISONS OF DISMASTD AND EXISTING METHODS

Method	Distributed	Tensor Type
BIGtensor [34]	✓	Static
SPLATT [16], [35]	✓	Static
CDTF & SALS [15]	✓	Static
DisTenC [12]	✓	Static
CartHP [36]	✓	Static
DBTF [37], [38]	✓	Static
PARAFAC-SDT & PARAFAC-RLST [39]	×	Traditional streaming
OnlineCP [18]	×	Traditional streaming
ITA [41], [42]	×	Traditional streaming
ALTO [43]	×	Traditional streaming
MAST [20]	×	Multi-aspect streaming
SIITA [19]	×	Multi-aspect streaming
DisMASTD	✓	**Multi-aspect streaming**

scalable imputation of incomplete streaming tensors. Zhou et al. [18] develop OnlineCP, an online algorithm that can incrementally track CP decomposition of dynamic tensors with arbitrary modes. In addition of CP decomposition, there are also Tucker decomposition methods. Sun et al. [41], [42] present an Incremental Tensor Analysis (ITA) framework to solve general streaming tensor analysis. Yu et al. [43] propose an accelerated online low-rank tensor learning algorithm (ALTO) for streaming tensor Tucker factorization. Some online Tucker decomposition approaches aim at not only one mode increasing setting [44], [45], but also deriving possible solutions for multi-aspect streaming using matrix-based methods, e.g., incremental SVD [46]. Recently, Song et al. [20] formally define the problem of multi-aspect streaming tensor completion, and present a CP-based algorithm (MAST) as a solution. Nimishakavi et al. [19] further explore the problem of multi-aspect streaming tensor completion with side information (SIITA).

Table I lists a comparison of DisMASTD and existing methods in terms of whether a method can support distributed environment and which tensor type it is designed for. Almost all the listed methods can only support the *static* or *traditional* streaming tensor decomposition. MAST and SIITA, the only two methods suitable for multi-aspect streaming tensor decomposition, are in the *centralized environment*. Our proposed DisMASTD in this paper is the first attempt to tackle *distributed* multi-aspect streaming tensor decomposition, which is expected to be able to cope with consistent data growth.

III. PRELIMINARIES

In this section, we formally give the definitions related to multi-aspect streaming tensor decomposition. Table II summarizes the symbols used frequently throughout this paper.

A. Tensor Notations

First, we introduce the notations, definitions, and basic operations related to multi-aspect streaming tensor decomposition, following the symbols used in [1], [20].

Definition 1: **(Tensor).** An N^{th}-order tensor is an N-way array, also known as an N-dimensional or N-mode tensor, denoted by $\mathcal{X}$, where $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$.

TABLE II
SYMBOLS AND DESCRIPTION

Notation	Description
$\mathcal{X}$	a tensor (Euler script letter)
$\widetilde{\mathcal{X}}$	the previous snapshot tensor of $\mathcal{X}$
N	the order of tensor
$\mathcal{X} \backslash \widetilde{\mathcal{X}}$	the relative complement of $\widetilde{\mathcal{X}}$ in $\mathcal{X}$
$\mathcal{X}[i_1^t, \ldots, i_N^t]$	an element of $\mathcal{X}$ indexed by $[i_1^t, \ldots, i_N^t]$
$\mathbf{X}_{(n)}$	the mode-n unfolding of $\mathcal{X}$
$nnz(\cdot)$	the number of non-zero elements, e.g., $nnz(\mathcal{X})$
$\mathbf{A}$	a matrix (boldface capital letter)
$\mathbf{A}^\intercal$	the transpose of $\mathbf{A}$
$[\![\cdot]\!]$	Kruskal operator, e.g., $\mathcal{X} \approx [\![\mathbf{A}_1, \ldots, \mathbf{A}_N]\!]$
$\|\cdot\|_F^2$	Frobenius norm, e.g., $\|\mathbf{A}\|_F^2$
$\odot$	Khatri-Rao product
$*$	Hadamard product
$(\mathbf{A}_k)^{\odot k \neq n}$	$\mathbf{A}_N \odot \ldots \odot \mathbf{A}_{n+1} \odot \mathbf{A}_{n-1} \odot \ldots \odot \mathbf{A}_1$
$(\mathbf{A}_k)^{* k \neq n}$	$\mathbf{A}_N * \ldots * \mathbf{A}_{n+1} * \mathbf{A}_{n-1} * \ldots * \mathbf{A}_1$

We utilize the term *order* to denote the dimensionality of a tensor (e.g., an N^{th}-order tensor), and the term *mode* to refer to a specific dimension of a tensor (e.g., the mode-n of tensor). Matrices and tensors are denoted by boldface capital letters (e.g., $\mathbf{A}$) and Euler script letters (e.g. $\mathcal{X}$), respectively. An element of a tensor $\mathcal{X}$ indexed by $[i_1^t, \ldots, i_N^t]$ is denoted as $\mathcal{X}[i_1^t, \ldots, i_N^t]$. To denote a varying index, we use the colon. For example, $\mathbf{A}[i, :]$ denotes the i-th row of matrix $\mathbf{A}$.

Definition 2: **(Tensor Matricization).** A tensor can be unfolded, or matricized, into a matrix by any of its modes. The mode-n unfolding of $\mathcal{X}$, denoted as $\mathbf{X}_{(n)}$, arranges the mode-n fibers to be the columns of the resulting matrix.

If $\mathcal{X}$ is of dimension $I \times J \times K$, then $\mathbf{X}_{(1)}$ is of dimension $I \times JK$. Note that, $\mathbf{A}^\intercal$ and $\|\mathbf{A}\|_F^2$ denote the transpose and Frobenius norm of matrix $\mathbf{A}$, respectively. Notations $[\![\cdot]\!]$, $\odot$, and $*$ represent the Kruskal operator, Khatri-Rao product, and Hadamard product, respectively.

Definition 3: **(CP Decomposition).** Given an N^{th}-order tensor $\mathcal{X}$, CP decomposition is an approximation of N factor matrices $\mathbf{A}_n \in \mathbb{R}^{I_n \times R}$, $n = 1, \ldots, N$, such that

$$\mathcal{X} \approx [\![\mathbf{A}_1, \ldots, \mathbf{A}_N]\!].$$

where R is usually a small positive integer denoting an upper bound of the rank of $\mathcal{X}$.

CP decomposition could be further written in matricized form as follows:

$$\mathcal{X} \overset{\text{unfold}}{\Longrightarrow} \quad \mathbf{X}_{(n)} \approx \mathbf{A}_n(\mathbf{A}_N \odot \ldots \mathbf{A}_{n+1} \odot \mathbf{A}_{n-1} \ldots \odot \mathbf{A}_1)^\intercal$$
$$= \mathbf{A}_n[(\mathbf{A}_k)^{\odot k \neq n}]^\intercal.$$

Definition 4: **(Multi-Aspect Streaming Tensor Sequence [20]).** A sequence of N^{th}-order tensors $\{\mathcal{X}^{(T)}\}$ is called multi-aspect streaming tensor sequence if $\forall T \in \mathbb{Z}^+$, $\mathcal{X}^{(T-1)}$ is the sub-tensor of $\mathcal{X}^{(T)}$, denoted as $\mathcal{X}^{(T-1)} \subseteq \mathcal{X}^{(T)}$. T increases with time, and $\mathcal{X}^{(T)}$ is the snapshot tensor of this sequence taken at time T.

Based on the definitions above, the definition of multi-aspect streaming tensor decomposition is formalized in Definition 5 below.

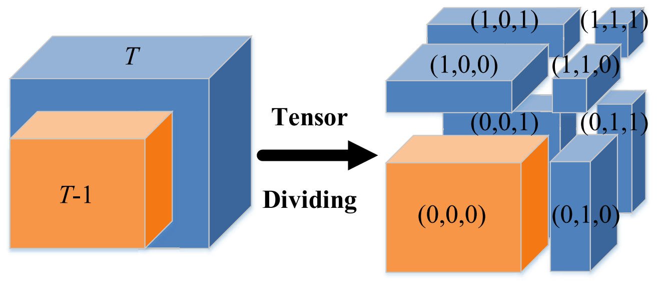


Fig. 2. Illustration of tensor division

Definition 5: **(Multi-Aspect Streaming Tensor Decomposition, MASTD [20]).** Given a multi-aspect streaming tensor sequence $\{\mathcal{X}^{(T)}\}$, Multi-Aspect Streaming Tensor Decomposition (MASTD) aims at decomposing the tensor in current snapshot $\mathcal{X}^{(T)}$ based on decomposition result corresponding to the tensor $\mathcal{X}^{(T-1)}$ in the previous time step.

B. Dynamic Tensor Decomposition

Towards efficient MASTD, a dynamic CP-based decomposition method DTD is proposed [20]. To simplify the discussion on DTD, Song et al. [20] focus on third-order multi-aspect streaming tensor decomposition. We will firstly introduce DTD in third-order tensors, and then, we extend it to general N^{th}-order tensors.

Given two consecutive snapshot third-order tensors $\mathcal{X}^{(T-1)}$ and $\mathcal{X}^{(T)}$ ($T \in \mathbb{Z}^+$), we denote them as $\widetilde{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times I_3}$ and $\mathcal{X} \in \mathbb{R}^{(I_1+d_1)\times(I_2+d_2)\times(I_3+d_3)}$, respectively. Let $\mathbf{A} \in \mathbb{R}^{(I_1+d_1)\times R}$, $\mathbf{B} \in \mathbb{R}^{(I_2+d_2)\times R}$, and $\mathbf{C} \in \mathbb{R}^{(I_3+d_3)\times R}$ be the CP factor matrices of tensor $\mathcal{X}$. CP decomposition solves the following optimization problem,

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\text{minimize}} \quad \mathcal{L}(\mathbf{A},\mathbf{B},\mathbf{C}) = \|\mathcal{X} - [\![\mathbf{A},\mathbf{B},\mathbf{C}]\!]\|_F^2. \tag{1}$$

The third-order tensor $\mathcal{X}$ can be divided into eight sub-tensors according to $\widetilde{\mathcal{X}}$ (as shown in Fig. 2). Each of those eight sub-tensors could be denoted by a binary tuple $(i,j,k) \in \{0,1\}^3 \triangleq \Theta$, with $\widetilde{\mathcal{X}} = \mathcal{X}^{0,0,0}$. Each pair of adjacent sub-tensors is coupled with each other. Based on the good division property of CP decomposition, each sub-tensor of $\mathcal{X}$ could also be approximated via the sub-matrices of $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$, i.e., $\mathcal{X}^{i,j,k} \approx [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]$, in which $\mathbf{A}_0 \in \mathbb{R}^{I_1\times R}$ and $\mathbf{A}_1 \in \mathbb{R}^{d_1\times R}$ are the divided sub-matrices of $\mathbf{A}$, i.e., $\mathbf{A}^\intercal = [\mathbf{A}_0^\intercal, \mathbf{A}_1^\intercal]$ (it applies to $\mathbf{B}_j$ and $\mathbf{C}_k$, too). Given the CP decomposition of previous snapshot $\widetilde{\mathcal{X}} \approx [\![\widetilde{\mathbf{A}},\widetilde{\mathbf{B}},\widetilde{\mathbf{C}}]\!]$, $\mathcal{X}^{0,0,0}$ could be approximated by the decomposition, and the loss function stated in the Equation (1) can be rewritten below,

$$\begin{aligned}\underset{\mathbf{A_i},\mathbf{B_j},\mathbf{C_k}}{\text{minimize}} \quad \mathcal{L}(\mathbf{A},\mathbf{B},\mathbf{C}) &= \sum_{(i,j,k)\in\Theta} \|\mathcal{X}^{i,j,k} - [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]\|_F^2 \\ &= \|\mathcal{X}^{0,0,0} - [\![\mathbf{A}_0,\mathbf{B}_0,\mathbf{C}_0]\!]\|_F^2 + \mathcal{L}_0 \\ &\approx \mu\|[\![\widetilde{\mathbf{A}},\widetilde{\mathbf{B}},\widetilde{\mathbf{C}}]\!] - [\![\mathbf{A}_0,\mathbf{B}_0,\mathbf{C}_0]\!]\|_F^2 + \mathcal{L}_0.\end{aligned} \tag{2}$$

Here, $\mathcal{L}_0 \triangleq \sum_{(i,j,k)\in\Theta\setminus(0,0,0)} \|\mathcal{X}^{i,j,k} - [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]\|_F^2$, and weight μ is the forgetting factor [17] used to alleviate the influence of the previous decomposition error.

```
Algorithm 1: Dynamic Tensor Decomposition
Input: the current snapshot tensor $\mathcal{X}$, the CP
       decomposition $\{\widetilde{\mathbf{A}}_n\}_{n=1}^N$ of previous snapshot
       tensor $\widetilde{\mathcal{X}}$, the forgetting factor $\mu$
Output: the CP decomposition of current snapshot
        tensor $\{\mathbf{A}_n\}_{n=1}^N$
1 $\{\mathbf{A}_n^{(0)}\}_{n=1}^N \leftarrow \{\widetilde{\mathbf{A}}_n\}_{n=1}^N$
2 $\{\mathbf{A}_n^{(1)}\}_{n=1}^N \leftarrow \{\text{rand}(\mathbf{d_n}, R)\}_{n=1}^N$
3 repeat
4     for $n \leftarrow 1 : N$ do
5         update $\mathbf{A}_n^{(0)}$ and $\mathbf{A}_n^{(1)}$ by the update rules (5)
6     update the loss $\mathcal{L}$ using the Equation (4)
7 until fit ceases to improve or maximum iterations
8 return $\{\mathbf{A}_n\}_{n=1}^N$
```

The optimization problem defined in the Equation (2) can be solved by alternating least squares (ALS). The update rule for A_i has been derived in [20], as stated in the Equation (3).

$$\begin{aligned}\mathbf{A}_0 &\leftarrow \frac{\mu\widetilde{\mathbf{A}}[(\widetilde{\mathbf{C}}^\intercal\mathbf{C}_0) * (\widetilde{\mathbf{B}}^\intercal\mathbf{B}_0)] + \sum_{(j,k)\neq(0,0)} \mathbf{X}_{(1)}^{0,j,k}(\mathbf{C}_k \odot \mathbf{B}_j)}{(\sum_{k=0}^1 \mathbf{C}_k^\intercal\mathbf{C}_k) * (\sum_{j=0}^1 \mathbf{B}_j^\intercal\mathbf{B}_j) - (1-\mu)(\mathbf{C}_0^\intercal\mathbf{C}_0) * (\mathbf{B}_0^\intercal\mathbf{B}_0)}, \\ \mathbf{A}_1 &\leftarrow \frac{\sum_{j,k} \mathbf{X}_{(1)}^{1,j,k}(\mathbf{C}_k \odot \mathbf{B}_j)}{(\sum_{k=0}^1 \mathbf{C}_k^\intercal\mathbf{C}_k) * (\sum_{j=0}^1 \mathbf{B}_j^\intercal\mathbf{B}_j)}.\end{aligned} \tag{3}$$

Note that the same update rules are also applicable to matrices $\mathbf{B}_0$, $\mathbf{C}_0$, $\mathbf{B}_1$, and $\mathbf{C}_1$.

Next, we extend the update rules mentioned above to tensors of arbitrary orders. Given an N^{th}-order tensor $\mathcal{X}$, we generate and update factor matrices $\{\mathbf{A}_n^{(0)}\}$ and $\{\mathbf{A}_n^{(1)}\}(n = 1, 2, \ldots, N)$ to solve the following optimization problem,

$$\begin{aligned}\underset{\{\mathbf{A}_n\}_{n=1}^N}{\text{minimize}} \quad &\mathcal{L}(\mathbf{A}_1,\ldots,\mathbf{A}_N) \\ &= \mu\|[\![\widetilde{\mathbf{A}}_1,\ldots,\widetilde{\mathbf{A}}_N]\!] - [\![\mathbf{A}_0,\ldots,\mathbf{A}_N]\!]\|_F^2 + \mathcal{L}_0.\end{aligned} \tag{4}$$

in which,

$$\mathcal{L}_0 \triangleq \sum_{i\in S_n} \|\mathcal{X}^i - [\![\mathbf{A}_1,\ldots,\mathbf{A}_N]\!]\|_F^2,$$

$$S_n = \{(s_1,\ldots,s_N) \,|\, \sum_{k=1}^N s_k \neq 0, s_k \in \{0,1\}\}.$$

The corresponding update rules are as follows:

$$\begin{aligned}\mathbf{A}_n^{(0)} &\leftarrow \frac{\mu\widetilde{\mathbf{A}}_n[(\widetilde{\mathbf{A}}_k^\intercal\mathbf{A}_k^{(0)})^{*k\neq n}] + \sum_{i\in S_n^0} \mathbf{X}_{(n)}^i(\mathbf{A}_k^{(i_k)})^{\odot k\neq n}}{(\mathbf{A}_k^{(0)\intercal}\mathbf{A}_k^{(0)} + \mathbf{A}_k^{(1)\intercal}\mathbf{A}_k^{(1)})^{*k\neq n} - (1-\mu)(\mathbf{A}_k^{(0)\intercal}\mathbf{A}_k^{(0)})^{*k\neq n}}, \\ \mathbf{A}_n^{(1)} &\leftarrow \frac{\sum_{i\in S_n^1} \mathbf{X}_{(n)}^i(\mathbf{A}_k^{(i_k)})^{\odot k\neq n}}{(\mathbf{A}_k^{(0)\intercal}\mathbf{A}_k^{(0)} + \mathbf{A}_k^{(1)\intercal}\mathbf{A}_k^{(1)})^{*k\neq n}}.\end{aligned} \tag{5}$$

in which,

$$\begin{aligned}S_n^0 &= \{(s_1,\ldots,s_N) \,|\, \sum_{k=1}^N s_k \neq 0, s_k \in \{0,1\}, s_n = 0\}, \\ S_n^1 &= \{(s_1,\ldots,s_N) \,|\forall k \in \{1,\ldots,N\}, s_k \in \{0,1\}, s_n = 1\}.\end{aligned}$$

Algorithm 1 depicts the pseudo-code of our extended Dynamic Tensor Decomposition (DTD) algorithm for any N^{th}-order multi-aspect streaming tensor. It takes as inputs the current snapshot tensor $\mathcal{X}$, the CP decomposition $\{\widetilde{\mathbf{A}}_n\}_{n=1}^N$ of previous snapshot tensor $\widetilde{\mathcal{X}}$, i.e., $\widetilde{\mathcal{X}} \approx [\![\widetilde{\mathbf{A}}_1, \dots, \widetilde{\mathbf{A}}_N]\!]$, and the forgetting factor μ. First, DTD uses the CP decomposition $\{\widetilde{\mathbf{A}}_n\}_{n=1}^N$ of previous snapshot tensor $\widetilde{\mathcal{X}}$ to initialize $\{\mathbf{A}_n^{(0)}\}_{n=1}^N$ (line 1). Then, it randomly initializes the rest of factor matrices $\{\mathbf{A}_n^{(1)}\}_{n=1}^N$ (line 2). Next, it iteratively updates the $\{\mathbf{A}_n\}_{n=1}^N$ and the loss $\mathcal{L}$ (lines 3-6) until the convergence condition is satisfied (line 7). After the convergence, DTD outputs the CP decomposition of current snapshot tensor $\mathcal{X}$, i.e., $\{\mathbf{A}_n\}_{n=1}^N$, to stop the tensor decomposition (line 8).

IV. Our Proposed DisMASTD

In this section, we present the details of our proposed DisMASTD. DisMASTD contains two main parts, namely *data partitioning* and *distributed tensor decomposition*, as listed below:

- **Part 1: Data Partitioning.** In the first part, we try to partition large-scale multi-aspect streaming tensors and the corresponding factor matrices to ensure the load balancing among all the worker nodes in a distributed environment. We detail this part in Section IV-A.
- **Part 2: Distributed Tensor Decomposition.** In the second part, we focus on the computation of tensor decomposition with the objective to avoid repetitive computation and to reduce network communication. We describe this part in Section IV-B.

In addition, we analyze the complexity of DisMASTD in Section IV-C.

A. Data Partitioning

In order to ensure load balancing, which is the basic requirement of distributed environments, large-scale tensors should be partitioned to ensure each worker node in the distributed platform has an equal or close-to-equal workload. The existing medium-grain partitioning methods [16], [36] have achieved state-of-the-art performance in normal distributed tensor decomposition. Unfortunately, they cannot efficiently support the distributed multi-aspect streaming tensor decomposition, because none of them has considered the special characteristics of multi-aspect streaming tensors as well as they can only re-compute the tensor decomposition from sketch. In the following, we first introduce the special characteristics of multi-aspect streaming tensors, and then, we show how to utilize them to design data partitioning and to avoid the computation from the sketch.

1) Characteristics of Multi-Aspect Streaming Tensors: For ease of presentation, we again take the third-order multi-aspect streaming tensors as an example. Let's revisit the update rules for the factor matrices defined in the Equation (3). There are two important characteristics in the update rules. First, the previous snapshot tensor $\widetilde{\mathcal{X}} = \mathcal{X}^{0,0,0}$ is *independent* in the decomposition for the current snapshot tensor $\mathcal{X}$, since the CP decomposition of previous snapshot tensor could approximately replace it. In other words, $\widetilde{\mathcal{X}} \approx [\![\widetilde{\mathbf{A}}, \widetilde{\mathbf{B}}, \widetilde{\mathbf{C}}]\!]$ holds. We only need to focus on the relative complement of $\widetilde{\mathcal{X}}$ in $\mathcal{X}$, and denote it as $\mathcal{X}\backslash\widetilde{\mathcal{X}}$. Here, $\mathcal{X}\backslash\widetilde{\mathcal{X}} = \{\mathcal{X}^{i,j,k} | (i,j.k) \neq (0,0,0), (i,j,k) \in \{0,1\}^3 \triangleq \Theta\}$.

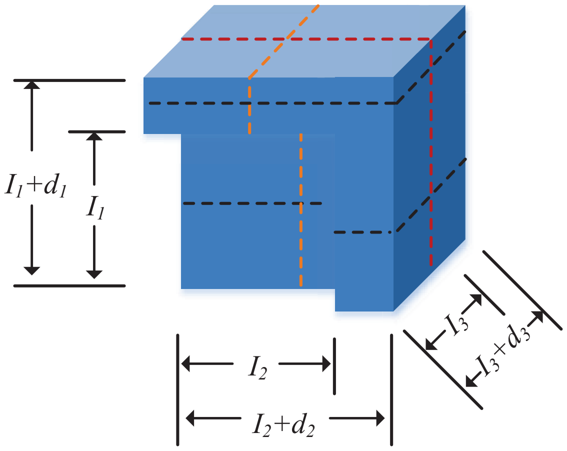


Fig. 3. Illustration of tensor partitioning

The second important characteristic is that there is an *expensive* operation in the Equation (3), which has the greatest potential to become the performance bottleneck. We denote $\hat{\mathbf{A}} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ as the Matricized Tensor Times Khatri-Rao Product (MTTKRP). The related terms in the Equation (3) include $\hat{\mathbf{A}}_0 = \sum_{(j,k)\neq(0,0)} \mathbf{X}_{(1)}^{0,j,k}(\mathbf{C}_k \odot \mathbf{B}_j)$ and $\hat{\mathbf{A}}_1 = \sum_{j,k} \mathbf{X}_{(1)}^{1,j,k}(\mathbf{C}_k \odot \mathbf{B}_j)$. Each element $\hat{\mathbf{A}}[i,f]$ could be computed as follows:

$$\hat{\mathbf{A}}[i,f] = \sum_{\mathcal{X}\backslash\widetilde{\mathcal{X}}[i,:,:]} \mathcal{X}\backslash\widetilde{\mathcal{X}}[i,j,k]\mathbf{B}[j,f]\mathbf{C}[k,f].$$

This element-wise computation lists two important properties of the MTTKRP operator. First, only the non-zero elements in $\mathcal{X}$ will contribute to the MTTKRP result. If an element in tensor is zero, the corresponding MTTKRP term is zero and hence will not contribute to the MTTKRP result. Second, the j and k indices in $\mathcal{X}$ determine the rows of factor matrices $\mathbf{B}$ and $\mathbf{C}$ that require accessing during the computation.

2) Tensor partitioning: The characteristics of multi-Aspect streaming tensors motivates us to derive a tensor partitioning such that it divides the tensor into partitions that contain equal-number non-zero tensor elements. This could achieve the optimal load balancing tensor partitioning. Fig. 3 illustrates an example for a third-order multi-aspect streaming tensor partitioning, in which black, orange, and red dashed lines depict the partitioning boundaries for the first, second, and third modes of tensor, respectively. However, the optimal tensor partitioning problem is NP-hard. In the following, we first prove this in Theorem 1, and then, we propose two heuristic-based tensor partitioning approaches.

Theorem 1: The optimal tensor partitioning problem is NP-hard.

Proof 1: We prove this by reducing our optimal tensor partitioning problem to the Partition problem [47], which is

NP-complete. In number theory and computer science, the Partition problem is the task of deciding whether a given multiset S of n positive integers, i.e., $S = \{s_i | i = 1, 2, ..., n, s_i \in \mathbb{Z}^+\}$, can be partitioned into two subsets S_1 and S_2 such that $\sum_{s_i \in S_1} s_i = \sum_{s_j \in S_2} s_j$, where $S_1 \cap S_2 = \emptyset$ and $S_1 \cup S_2 = S$. We are able to reduce the optimal tensor partitioning problem as follows:

First, we could set the partition number p to 2, meaning that we divide the given tensor into 2 parts such that the optimal load balancing could be achieved. Second, we assume that the mode to be split is the first mode (otherwise, we could traverse all the modes to find the eligible mode in linear time), and denote the number of non-zero elements in i-th slice of the first-mode tensor as $a_i = nnz(\mathcal{X} \backslash \widetilde{\mathcal{X}}[i, :, :])$, $i = 1, 2, \ldots, I$.

The workload of each partition depends on the number of non-zero elements in the partition. Thus, the solution of the optimal load balancing tensor partitioning is to derive two partitions P_1 and P_2 of $\mathcal{X} \backslash \widetilde{\mathcal{X}}$ such that $\sum_{a_i \in P_1} a_i = \sum_{a_i \in P_2} a_i$, $P_1 \cap P_2 = \emptyset$ and $P_1 \cup P_2 = \mathcal{X} \backslash \widetilde{\mathcal{X}}$. This is equivalent to the Partition problem. Therefore, the optimal tensor partitioning problem is NP-hard. The proof completes. ■

Due to the hardness of the optimal tensor partitioning problem, we utilize two heuristic-based approaches to perform the partitioning, namely, *Greedy Tensor Partitioning* (*GTP*) and *Max-min Tensor Partitioning* (*MTP*), as detailed below.

Greedy Tensor Partitioning. GTP assigns partition boundaries greedily in each mode. To be more specific, GTP first partitions the tensor in the first mode. It greedily assigns partition boundaries such that the $nnz(\mathcal{X} \backslash \widetilde{\mathcal{X}})$ of each partition in the first mode reaches the target size $nnz(\mathcal{X} \backslash \widetilde{\mathcal{X}}))/q$. Note that, the target size is an optimal set such that each partition shares the same number of non-zero elements. The same strategy could be applied to every other mode.

Algorithm 2 presents the corresponding pseudo-code of GTP. It takes as inputs the relative complement tensor $\mathcal{X} \backslash \widetilde{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and the number of partitions in each mode $\{p_n\}_{n=1}^N$, and outputs the partition results $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$. The main body of the algorithm is a for-loop, which greedily assigns slices to the partitions for each mode of the given tensor (lines 1-17). For a given mode n (line 1), it first computes the target ω, i.e., the optimal number of non-zero elements (i.e., nnz, which denotes the number of non-zero elements) in each partition (line 2). Then, it computes and stores nnz in the i-th slice in the current mode to $a_i^{(n)}$ (line 3). Meanwhile, it initializes the container $\{P_p^{(n)}\}_{p=1}^{p_n}$ to store partition results, as well as other temporary container variables (lines 4-5). Next, GTP greedily assigns the slices to the current partition, i.e., the temporary variable P, until the total number of non-zero elements in the current partition (captured by variable sum) is no smaller than the target ω (lines 6-9). Note that, slices could have skewed density of non-zero elements. When GTP adds a slice i with large nnz to the current partition, the nnz in the partition could be significantly larger than the target ω. Thus, if the current sum is larger than ω (line 10), GTP compares the partition without slice i against

Algorithm 2: Greedy Tensor Partitioning

```
Input: the relative tensor complement
       $\mathcal{X} \backslash \widetilde{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$, the number of
       partitions in each mode $\{p_n\}_{n=1}^N$
Output: the partition results $\{P_p^{(n)}\}_{p=1}^{p_n}$, $n = 1, 2, \ldots, N$
1  for $n \leftarrow 1 : N$ do
2      $\omega \leftarrow nnz(\mathcal{X} \backslash \widetilde{\mathcal{X}})/p_n$
3      $\{a_i^{(n)}\}_{i=1}^{I_n} \leftarrow$ the $nnz$ of each slice in mode $n$
4      $\{P_p^{(n)}\}_{p=1}^{p_n} \leftarrow \{\emptyset\}$
5      $P \leftarrow \emptyset$, $count \leftarrow 0$, $sum \leftarrow 0$
6      for $i \leftarrow 1 : I_n$ do
7          $sum \leftarrow sum + a_i^{(n)}.nnz$
8          if $sum < \omega$ then
9              assign slice $i$ to $P$
10         else                                  // $sum \geq \omega$
11             if $sum - \omega < \omega - (sum - a_i^{(n)}.nnz)$ then
12                 assign slice $i$ to $P$
13             if $count < p_n$ then
                   // form a new partition
14                 $P_{count}^{(n)} \leftarrow P$
15                 $P \leftarrow \emptyset$, $count \leftarrow count + 1$, $sum \leftarrow 0$
16             else
17                 assign remaining slices to $P_{p_n}^{(n)}$, break
18 return $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$
```

the partition having included slice i, and chooses whichever that leads to a better balance (lines 11-12). After that, GTP generates a partition $P_{count}^{(n)}$ for mode n, and proceeds to the next partition (lines 13-15) until p_n partitions are generated. Note that, when the number of partitions reaches p_n, GTP stops the evaluation, and adds all the remaining slices to the last partition (i.e., the p_n-th partition in mode n) to end the partitioning process corresponding to the current mode n (lines 16-17). After all modes are partitioned, GTP outputs the tensor partitioning results $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$ (line 18).

Max-min Fit Tensor Partitioning. GTP follows the original order of the slices when grouping slices into partitions. Considering the skewed non-zero element distribution in tensors, GTP might not be able to achieve a balanced partitioning. Motivated by this observation, we present MTP algorithm. Unlike GTP, MTP first sorts the slices by the number of non-zero elements in each slice, and then, it assigns the slice with the current biggest nnz to the partition with the current smallest nnz.

Algorithm 3 shows the pseudo-code of MTP. It shares the same inputs and same output as GTP. MTP partitions the tensor mode by mode in the for-loop (lines 1-7). For a given mode n (line 1), it first computes the nnz $\{a_i^{(n)}\}_{i=1}^{I_n}$ in each slice (line 2), and sorts the slices according to descending

Algorithm 3: Max-min Fit Tensor Partitioning

```
Input: the relative tensor complement
       $\mathcal{X}\backslash\widetilde{\mathcal{X}} \in \mathbb{R}^{I_1\times I_2\times\ldots\times I_N}$, the number of
       partitions in each mode $\{p_n\}_{n=1}^{N}$
Output: the partition results
        $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$
1 for $n \leftarrow 1 : N$ do
2     $\{a_i^{(n)}\}_{i=1}^{I_n} \leftarrow$ the $nnz$ of each slice in mode $n$
3     sort $\{a_i^{(n)}\}$ based on descending order of $nnz$
4     $\{P_p^{(n)}\}_{p=1}^{p_n} \leftarrow \{\emptyset\}$
5     for $i \leftarrow 1 : I_n$ do
6         $p \leftarrow P_p^{(n)}$ in $\{P_p^{(n)}\}$ with the minimal $nnz$
            sum
7         assign $a_i^{(n)}$ to $p$
8 return $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$
```

order of their nnz values (i.e., $\forall 1 \leq j < k \leq I_n$, $a_j^{(n)} \geq a_k^{(n)}$, line 3). Then, it initializes the container $\{P_p^{(n)}\}_{p=1}^{p_n}$ to store partition results (line 4). Next, MTP iteratively assigns the current slice that contains at least as many non-zero elements as any of the remaining slices to a partition with the smallest number of non-zero elements (lines 5-7). The partition of one mode is completed after all $a_i^{(n)}$s are assigned to different partitions. After all the N modes are partitioned, MTP outputs the tensor partitioning results $\{P_p^{(n)}\}_{p=1}^{p_n}, n = 1, 2, \ldots, N$ (line 8).

3) Factor Matrix Partitioning: After the tensor partitioning, we induce the partitions for the factor matrices based on the non-zero elements in tensor partitions. Let's revisit the Equation (6). The indices of non-zero tensor elements determine the related rows of factor matrices that need to be accessed during the MTTKRP computation. Thus, we assign all the related factor matrices to the corresponding tensor partitions in a row-wise pattern. Fig. 4 illustrates a tensor partition (shown as the cube), with its corresponding rows of factor matrices (shown as the rectangles). Note that, more details about Fig. 4 will be stated in Section IV-B1.

B. Distributed Tensor Decomposition

After data partitioning, each partition holds a non-zero tensor as well as the related rows of factor matrices. In the following, we detail each computation step of distributed tensor decomposition. For brevity, we consider the computation corresponding to the first mode of a third-order tensor, although the computations of other modes or even the computations of more general order tensors could be extended and performed in a similar manner.

1) Distributed MTTKRP Computation: In each partition, for each non-zero element in $\mathcal{X}\backslash\widetilde{\mathcal{X}}$, we first perform MTTKRP computation for the factor matrix $\mathbf{A}$ as follows:

$$\hat{\mathbf{A}}[i,:] = \sum_{\mathcal{X}\backslash\widetilde{\mathcal{X}}[i,:,:]} \mathcal{X}\backslash\widetilde{\mathcal{X}}[i,j,k]\mathbf{B}[j,:]\mathbf{C}[k,:]. \quad (6)$$

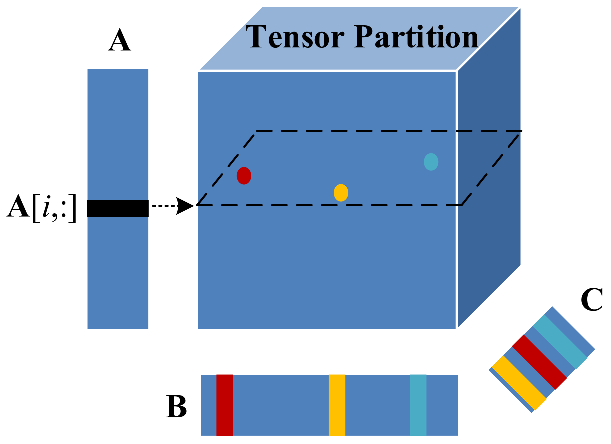


Fig. 4. Illustration of a tensor partition and the corresponding rows of factor matrices

This computation could be done at the granularity of factor matrix rows. Consider the example in Fig. 4 again. For the row $\mathbf{A}[i,:]$ of factor matrix $\mathbf{A}$, the MTTKRP computation needs to access every non-zero element $\mathcal{X}\backslash\widetilde{\mathcal{X}}[i,:,:]$ from the i-th slice of the first mode in tensor $\mathcal{X}\backslash\widetilde{\mathcal{X}}$, shown as the dots in the area bounded by dashed-lines in Fig. 4. In addition, it needs to access every row $\mathbf{B}[j,:]$ and every row $\mathbf{C}[k,:]$ in factor matrices $\mathbf{B}$ and $\mathbf{C}$ indexed by every non-zero element $\mathcal{X}\backslash\widetilde{\mathcal{X}}[i,j,k]$ in the i-th tensor slice $\mathcal{X}\backslash\widetilde{\mathcal{X}}[i,:,:]$, depicted as strips in Fig. 4. Finally, all the MTTKRP partial terms are aggregated to get the result $\hat{\mathbf{A}}[i,:]$.

2) Distributed Factor Matrix Update: After the distributed MTTKRP computation, the intermediate results should be processed with other factor matrices to update the factor matrices shown as the update rules defined in the Equation (3). In the following, we first introduce the update of factor matrix $\mathbf{A}_0$, and then consider the update of factor matrix $\mathbf{A}_1$.

For the factor matrix $\mathbf{A}_0$, the related factor matrices include $\widetilde{\mathbf{A}}$, $\widetilde{\mathbf{B}}$, $\widetilde{\mathbf{C}}$, $\mathbf{B}_0$, $\mathbf{B}_1$, $\mathbf{C}_0$, and $\mathbf{C}_1$. To be more specific, given a local row $\mathbf{A}_0[i,:]$ of the factor matrix $\mathbf{A}_0$, we need to access the corresponding $\widetilde{\mathbf{A}}[i,:]$ row of $\widetilde{\mathbf{A}}$, $\widetilde{\mathbf{C}}^\intercal\mathbf{C}_0$, $\widetilde{\mathbf{B}}^\intercal\mathbf{B}_0$, $\mathbf{C}_k^\intercal\mathbf{C}_k$ ($k = 0, 1$), and $\mathbf{B}_j^\intercal\mathbf{B}_j$ ($j = 0, 1$) for the update. Among these matrices, the products of factor matrices are $R \times R$ matrices, which are small enough to be fit into the memory of each partition. Here, we assume that they are cached in every partition. Besides, all the related rows of factor matrices have been already cached in each partition. Thus, we could perform the update rule of $\mathbf{A}_0$ in row-wise distribution pattern, as detailed below.

$$\mathbf{A}_0[i,:] \leftarrow \frac{\mu\widetilde{\mathbf{A}}[i,:][(\widetilde{\mathbf{C}}^\intercal\mathbf{C}_0) * (\widetilde{\mathbf{B}}^\intercal\mathbf{B}_0)] + \hat{\mathbf{A}}[i,:]}{(\sum_{k=0}^{1}\mathbf{C}_k^\intercal\mathbf{C}_k) * (\sum_{j=0}^{1}\mathbf{B}_j^\intercal\mathbf{B}_j) - (1-\mu)(\mathbf{C}_0^\intercal\mathbf{C}_0) * (\mathbf{B}_0^\intercal\mathbf{B}_0)}.$$

The update of factor matrix $\mathbf{A}_1$ is similar to that of factor matrix $\mathbf{A}_0$. Moreover, the factor matrix $\mathbf{A}_1$ is independent on the previous snapshot factor matrices (i.e., $\widetilde{\mathbf{A}}, \widetilde{\mathbf{B}}, \widetilde{\mathbf{C}}$). Thus, the update of $\mathbf{A}_1$ is even simpler than that of $\mathbf{A}_0$. The corresponding update rule of $\mathbf{A}_1$ is as follows:

$$\mathbf{A}_1[i,:] \leftarrow \frac{\hat{\mathbf{A}}[i,:]}{(\sum_{k=0}^{1}\mathbf{C}_k^\intercal\mathbf{C}_k) * (\sum_{j=0}^{1}\mathbf{B}_j^\intercal\mathbf{B}_j)}.$$

3) Distributed Matrix Product Update: Since we assume that all the products of matrices are cached when updating the factor matrix, each partition needs to compute the products of matrices for the next iteration round.

Given a matrix product $\mathbf{A}^\intercal\mathbf{B}$, we could compute it in a row-wise form:

$$\mathbf{A}^\intercal\mathbf{B} = \begin{bmatrix} \mathbf{A}_{P_1}^\intercal & \mathbf{A}_{p_2}^\intercal & \dots & \mathbf{A}_{P_p}^\intercal \end{bmatrix} \begin{bmatrix} \mathbf{B}_{P_1} \\ \mathbf{B}_{P_2} \\ \dots \\ \mathbf{B}_{P_p} \end{bmatrix} = \sum_{i=1}^{p} \mathbf{A}_{P_i}^\intercal\mathbf{B}_{P_i}.$$

After each partition updates the corresponding rows of factor matrix $\mathbf{A}_0[i,:]$, it first computes its local $\mathbf{A}_0[i,:]^\intercal\mathbf{A}_0[i,:]$ and $\widetilde{\mathbf{A}}[i,:]^\intercal\mathbf{A}_0[i,:]$. Then, it performs an All-to-All reduction to aggregate the final matrices $\widetilde{\mathbf{A}}^\intercal\mathbf{A}_0$ and $\mathbf{A}_0^\intercal\mathbf{A}_0$, and distributes them among all partitions.

As for the factor matrix $\mathbf{A}_1$, we could follow the above process to generate the matrix product $\mathbf{A}_1^\intercal\mathbf{A}_1$ for the next iteration round.

4) Distributed Loss Function Computation: The the loss function should be computed at the end of each iteration round for the convergence condition. As stated in the Equation (2), the loss function could be divided into two parts:

$$\begin{aligned}\mathcal{L} &= \mathcal{L}^{(0,0,0)} + \sum_{(i,j,k)\in\Theta\backslash(0,0,0)} \mathcal{L}^{(i,j,k)},\\ \mathcal{L}^{(0,0,0)} &\triangleq \mu\|[\![\widetilde{\mathbf{A}},\widetilde{\mathbf{B}},\widetilde{\mathbf{C}}]\!] - [\![\mathbf{A}_0,\mathbf{B}_0,\mathbf{C}_0]\!]\|_F^2,\\ \mathcal{L}^{(i,j,k)} &\triangleq \|\mathcal{X}^{i,j,k} - [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]\|_F^2.\end{aligned}$$

where weight μ is the forgetting factor.

In the following, we first discuss the computation of the first part $\mathcal{L}^{(0,0,0)}$, and then detail the computation of the second part $\mathcal{L}^{(i,j,k)}$ $((i,j,k)\in\Theta\backslash(0,0,0))$.

The first part of loss function $\mathcal{L}^{(0,0,0)}$ could be rewritten as:

$$\begin{aligned}\mathcal{L}^{(0,0,0)} &= \mu\|[\![\widetilde{\mathbf{A}},\widetilde{\mathbf{B}},\widetilde{\mathbf{C}}]\!] - [\![\mathbf{A}_0,\mathbf{B}_0,\mathbf{C}_0]\!]\|_F^2,\\ &= \mu(\|\widetilde{\mathbf{A}}^\intercal\widetilde{\mathbf{A}} * \widetilde{\mathbf{B}}^\intercal\widetilde{\mathbf{B}} * \widetilde{\mathbf{C}}^\intercal\widetilde{\mathbf{C}}\|_F^2 + \|\mathbf{A}_0^\intercal\mathbf{A}_0 * \mathbf{B}_0^\intercal\mathbf{B}_0 * \mathbf{C}_0^\intercal\mathbf{C}_0\|_F^2\\ &\quad - 2\|\widetilde{\mathbf{A}}^\intercal\mathbf{A}_0 * \widetilde{\mathbf{B}}^\intercal\mathbf{B}_0 * \widetilde{\mathbf{C}}^\intercal\mathbf{C}_0\|_F^2).\end{aligned}$$

The first term $\|\widetilde{\mathbf{A}}^\intercal\widetilde{\mathbf{A}} * \widetilde{\mathbf{B}}^\intercal\widetilde{\mathbf{B}} * \widetilde{\mathbf{C}}^\intercal\widetilde{\mathbf{C}}\|_F^2$ is a constant, i.e., it does not change its value in the iteration. These factor matrices of the previous snapshot including $\widetilde{\mathbf{A}}$, $\widetilde{\mathbf{B}}$, and $\widetilde{\mathbf{C}}$ are constant inputs, and thus, $\|\widetilde{\mathbf{A}}^\intercal\widetilde{\mathbf{A}} * \widetilde{\mathbf{B}}^\intercal\widetilde{\mathbf{B}} * \widetilde{\mathbf{C}}^\intercal\widetilde{\mathbf{C}}\|_F^2$ could be pre-computed. The other terms of $\mathcal{L}^{(0,0,0)}$ can be computed directly, as the products of factor matrices $\mathbf{A}_0^\intercal\mathbf{A}_0$ and $\widetilde{\mathbf{A}}^\intercal\mathbf{A}_0$ (similar to $\mathbf{B}$ and $\mathbf{C}$) are computed during the iteration, as stated in the above discussion. We could maintain and reuse these intermediate results.

The second part of the loss function could be computed as follows. Given a tuple $(i,j,k)\in\Theta\backslash(0,0,0)$, we aim to compute the partial loss $\mathcal{L}^{(i,j,k)} = \|\mathcal{X}^{i,j,k} - [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]\|_F^2$. We denote the approximation of $\mathcal{X}^{i,j,k}$ as $\mathcal{Y}^{i,j,k}$, i.e. $\mathcal{Y}^{i,j,k} = [\![\mathbf{A}_i,\mathbf{B}_j,\mathbf{C}_k]\!]$. Then, the partial loss $\mathcal{L}^{(i,j,k)}$ can be represented as:

$$\sqrt{\langle\mathcal{X}^{i,j,k},\mathcal{X}^{i,j,k}\rangle + \langle\mathcal{Y}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle - 2\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle}.$$

The first term $\langle\mathcal{X}^{i,j,k},\mathcal{X}^{i,j,k}\rangle = \|\mathcal{X}^{i,j,k}\|_F^2$ is the tensor norm, which is the square root of the sum of the squares of all its non-zero elements. $\mathcal{X}^{i,j,k}$ is also a constant input, and hence, its norm can be pre-computed.

The second norm of the approximate tensor $\mathcal{Y}^{i,j,k}$ can be represented as:

$$\langle\mathcal{Y}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle = \|\mathcal{Y}^{i,j,k}\|_F^2 = \|\mathbf{A}_i^\intercal\mathbf{A}_i * \mathbf{B}_j^\intercal\mathbf{B}_j * \mathbf{C}_k^\intercal\mathbf{C}_k\|_F^2.$$

Again, the norm can be computed by using the intermediate results. This is because the matrix products are computed during the iteration.

The last term, i.e., the inner product term $\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle$, could be represented as follows:

$$\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle = \sum_{\text{nnz}(\mathcal{X})} \mathcal{X}[i,j,k]\mathbf{A}[i,f]\mathbf{B}[j,f]\mathbf{C}[k,f]. \quad (7)$$

Before we explain the detailed calculation of this term, let's revisit the MTTKRP computation defined in the Equation (6). We could observe that the partial product between $\mathbf{B}$, $\mathbf{C}$, and $\mathcal{X}$ is already computed as $\hat{\mathbf{A}}$. In other words, we have $\hat{\mathbf{A}}[i,f] = \sum_{\text{nnz}(\mathcal{X})}\mathcal{X}[i,j,k]\mathbf{B}[j,f]\mathbf{C}[k,f]$. Inspired by this, we could maintain $\hat{\mathbf{A}}$, and rewrite the Equation (7) in a row-wise form, and then, we aggregate to get the inner product term $\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle$, as follows:

$$\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle = \sum_i \hat{\mathbf{A}}[i,:]^\intercal\mathbf{A}[i,:].$$

In other words, we could reuse the row of factor matrix and its corresponding MTTKRP result to derive the inner product in a row-wise form and to aggregate the inner product term $\langle\mathcal{X}^{i,j,k},\mathcal{Y}^{i,j,k}\rangle$.

So far, we have detailed each and every step to compute the distributed multi-aspect streaming tensor decomposition. DisMASTD iterates the above computation steps until the convergence.

C. Complexity Analysis

In this subsection, we theoretically analyze our proposed DisMASTD, including time complexity, memory requirement, and network communication, in Theorem 2, Theorem 3, and Theorem 4, respectively.

For the sake of simplicity, we assume that notations $\widetilde{\mathcal{X}} \in \mathbb{R}^{I\times I\times\ldots\times I}$, $\mathcal{X}\in\mathbb{R}^{(I+d)\times(I+d)\times\ldots\times(I+d)}$ represent the two N^{th}-order consecutive snapshots tensors, notation $\widetilde{\mathbf{A}}_i\in\mathbb{R}^{I\times R}$ refers to the previous snapshot factor matrix, and notation $\mathbf{A}_i\in\mathbb{R}^{(I+d)\times R}$, $i=1,2,\ldots,N$ stands for the current snapshot factor matrix, respectively. Let M be the number of worker nodes in a distributed environment. Note that, we take one iteration round of distributed tensor decomposition in DisMASTD as an example to analyze the complexity.

Theorem 2: The time complexity of our proposed DisMASTD is $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R + R^3 + IR^2 + dR^2 + IR +$

$dR + R^2 + I))$ when it uses GTP to partition tensors, or $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R+R^3+IR^2+dR^2+IR+dR+R^2+IlogI))$ when it uses MTP to partition tensors.

Proof 2: As stated in the beginning of Section IV, there are two main parts in DisMASTD, namely, data partitioning and distributed tensor decomposition.

In the data partitioning, DisMASTD firstly needs $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}))$ time to traverse the $\mathcal{X}\backslash\widetilde{\mathcal{X}}$, and compute nnz by slices in each mode. Then, it uses the statistics to derive the partition boundaries. For each mode, GTP takes $O(I)$ time in assigning partition boundaries, while MTP needs $O(IlogI)$ time. Since each non-zero tensor element is mapped to a partition with a constant time, mapping all the non-zero elements to corresponding partitions in each mode takes $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}))$ time. After the tensor partitioning, all the previous snapshot factor matrices $\widetilde{\mathbf{A}}_i$ are read in $O(NIR)$ time, and all the current snapshot factor matrices $\mathbf{A}_i$ are initialized in $O(NIR + NdR)$ time, $i = 1, 2, \ldots, N$. Meanwhile, all the factor matrices are assigned to corresponding partitions. Consequently, the data partitioning needs $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}) + I + IR + dR))$ time using GTP or $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}) + IlogI + IR + dR))$ time using MTP.

In the distributed tensor decomposition, computing MTTKRP in the granularity of matrix row for all the factor matrices takes $O(Nnnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R)$. Updating a factor matrix should compute the multiplication of two matrices, i.e., the numerator term and the inverse matrix of denominator term in the Equation (5). It requires $O(IR^2 + IR)$ time to get the numerator term; it needs $O(R^2)$ time to aggregates the matrix products to get the denominator term, and $O(R^3)$ time to inverse the denominator term. The multiplication of the two matrices takes $O((I + d)R^2)$ time. In total, it takes $O(N(R^3 + IR^2 + dR^2 + IR + R^2))$ time in updating all the factor matrices. Updating all the matrix products needs $O(N(I + d)R^2)$ time. Computing the convergence condition takes $O(NR^2 + (I + d)R^2)$ time, thanks to the reuse of the intermediate results. Overall, the distributed tensor decomposition part needs $O(Nnnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R + N(R^3 + IR^2 + dR^2 + IR + R^2) + N(I + d)R^2 + NR^2 + (I + d)R^2) = O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R + R^3 + IR^2 + dR^2 + IR + R^2))$ time.

By considering the costs corresponding to the two parts of DisMASTD, the total time complexity is $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R+ R^3 + IR^2 + dR^2 + IR + dR + R^2 + I))$ using GTP or $O(N(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})R+R^3+IR^2+dR^2+IR+dR+R^2+IlogI))$ using MTP. The proof completes. ■

Theorem 3: The memory requirement of our proposed DisMASTD is $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}) + MNR^2 + NIR + NdR)$.

Proof 3: DisMASTD needs to store following data items in memory to facilitate the decomposition of tensor: i) the tensor $\mathcal{X}\backslash\widetilde{\mathcal{X}}$; ii) the previous snapshot factor matrices $\widetilde{\mathbf{A}}_i$; iii) the current snapshot factor matrices $\mathbf{A}_i$; iv) the result of MTTKRP $\hat{\mathbf{A}}_i$; and v) the matrix products $\widetilde{\mathbf{A}}_i^{\intercal}\mathbf{A}_{i_0}$, $\mathbf{A}_{i_0}^{\intercal}\mathbf{A}_{i_0}$, $\mathbf{A}_{i_1}^{\intercal}\mathbf{A}_{i_1}$, $i = 1, 2, \ldots, N$. DisMASTD store $\mathcal{X}\backslash\widetilde{\mathcal{X}}$ by all the non-zero elements with the coordinate format in the distributed system. Thus, it needs $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}))$ memory to store the tensor $\mathcal{X}\backslash\widetilde{\mathcal{X}}$. Every factor matrix as well as the corresponding MTTKRP result are collectively owned by all the M worker nodes, which take $O(NIR + NdR)$ memory in total. In each mode, the matrix products require $O(R^2)$ memory. As DisMASTD broadcasts those matrix products among all the M worker nodes, the total memory requirement is $O(MNR^2)$. Hence, the amount of memory for storing data is $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}) + MNR^2 + NIR + NdR)$, which completes the proof. ■

Theorem 4: The network communication of our proposed DisMASTD is $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}) + MNR^2 + NIR + NdR)$.

Proof 4: In the data partitioning, it takes $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}}))$ network communication in mapping the non-zero elements to the corresponding partitions. All the factor matrices are mapped to corresponding partitions in the row-wise pattern, which takes $O(NIR + NdR)$ communication. In the distributed tensor decomposition, it needs $O(MNR^2)$ communication to aggregate and broadcost all those matrix products to M worker nodes. Besides, it takes $O(IR+dR)$ communication for the reuse of MTTKRP results to aggregate the loss. Thus, the total network communication is $O(nnz(\mathcal{X}\backslash\widetilde{\mathcal{X}})+MNR^2+ NIR + NdR)$. The proof completes. ■

TABLE III
STATISTICS OF THE DATASETS USED

Dataset	*I*	*J*	*K*	*nnz*
Clothing	1.2×10^7	2.7×10^6	7.0×10^3	3.2×10^7
Book	1.5×10^7	2.9×10^6	8.2×10^3	5.1×10^7
Netflix	4.8×10^5	1.8×10^4	2.2×10^3	1.0×10^8
Synthetic	5.0×10^4	5.0×10^4	5.0×10^4	5.0×10^8

V. Experimental Evaluation

In this section, we experimentally evaluate DisMASTD using both real and synthetic datasets, compared with the state-of-the-art competitor.

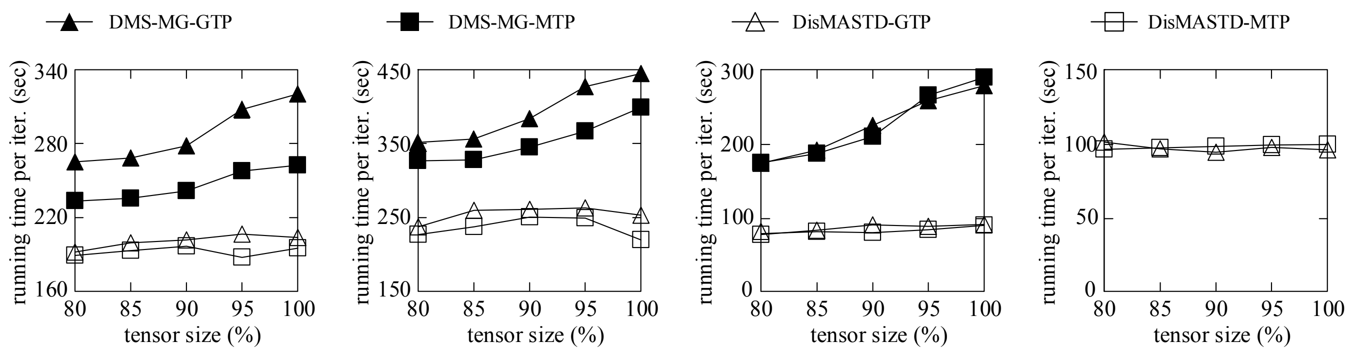
A. Experimental Setup

We employ three real datasets, i.e., *Clothing*, *Book*, and *Netflix* as well as one synthetic dataset *Synthetic*. Table III lists the statistics of the above-mentioned four datasets. Here, I, J, and K represent three different modes; and *nnz* is the number of non-zero elements in each dataset. *Clothing* and *Book* are two categories of reviews obtained from Amazon review dataset[2]. They can be parsed as two reviewer-product-time rating tensors. *Netflix* is a customer-movie-date rating tensor, which is taken from the Netflix Prize open competition[3]. Tensor decomposition in the three real datasets could be used for recommendation purpose as discussed in Section I. *Synthetic* is a third-order tensor, and its non-zero elements are set with uniform distribution.

We set the second dimension R of factor matrices to 10 and the forgetting factor μ to 0.8. Besides, we set the maximum

[2]https://nijianmo.github.io/amazon/index.html

[3]https://www.kaggle.com/netflix-inc/netflix-prize-data



(a) *Clothing* (b) *Book* (c) *Netflix* (d) *Synthetic*

Fig. 5. Running time per iteration versus the multi-aspect streaming tensor

TABLE IV
THE STANDARD DEVIATION STATICS OF *nnz* IN TENSOR PARTITIONS

Dataset	*p*	8	15	23	30	38
Clothing	GTP	0.7843	1.2743	1.6197	1.9361	2.1954
	MTP	0.0273	0.0716	0.1222	0.1720	0.2208
Book	GTP	0.9456	1.3321	1.6210	1.8491	2.1628
	MTP	0.0284	0.0617	0.1020	0.1496	0.2119
Netflix	GTP	0.4259	0.6496	1.1004	0.8861	0.9712
	MTP	0.0284	0.1095	0.2008	0.2457	0.2888
Synthetic	GTP	0.0047	0.0124	0.0240	0.0363	0.0503
	MTP	0.0048	0.0117	0.0223	0.0331	0.0473

number of iterations to 10, and report the average running time per iteration. Note that, we focus on the efficiency and scalability of tensor decomposition in this paper. The parameters mentioned above only have an impact on the accuracy of tensor decomposition, i.e., the similarity between the decomposition result and the origin tensor. We consistently set the parameters in all the experiments considering both accuracy and speed for fair comparison. All the experiments were implemented in Scala 2.11 on Spark 2.2, and run on a 15-node Dell cluster. Each node has two Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz processors with 12 cores, 128GB RAM, 1TB disk, and connect in Gigabit Ethernet.

B. Performance Study

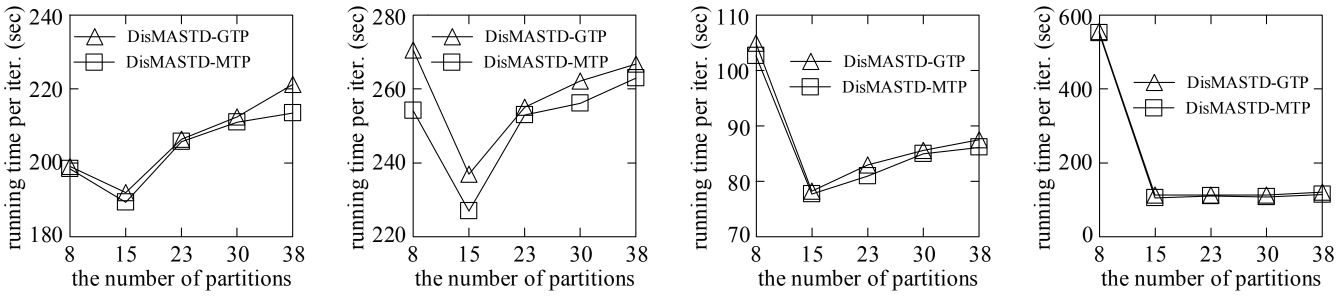
In this subsection, we evaluate the effectiveness and efficiency of our proposed DisMASTD. We first compare DisMASTD with the extended DMS-MG [16] on the multi-aspect streaming tensor in Section V-B1. Note that, DMS-MG is the state-of-the-art static tensor decomposition method. We extend DMS-MG to Spark using our framework, and implement two versions, including *DMS-MG-GTP* which uses GTP to partition tensors and *DMS-MG-MTP* which utilizes MTP to partition tensors. Then, we study the effect of the tensor partitioning and give the experimental guide to set the number of tensor partitions in Section V-B2. Finally, we conduct the experiments to verify the scalability of DisMASTD in Section V-B3.

1) Performance on the Multi-Aspect Streaming Tensor: We first study the efficiency of our proposed DisMASTD, including *DisMASTD-GTP* which uses GTP to partition tensors, and *DisMASTD-MTP* which utilizes MTP to partition tensors, compared against DMS-MG-GTP and DMS-MG-MTP on the multi-aspect streaming tensor.

Fig. 5 plots the average running time per iteration with respect to the multi-aspect streaming tensor, i.e., the tensor size increases from 75% to 100% of the whole dataset by 5% at each time step. We have made four observations in the following. First, DisMASTD performs much better than DMS-MG. Second, DisMASTD demonstrates significantly better scalability than DMS-MG. Furthermore, DMS-MG even fails to perform the tensor decomposition when the underlying dataset reaches certain size, e.g., *Synthetic* shown in Fig. 5(d). This is because DisMASTD benefits from an important characteristic of multi-aspect streaming tensors, i.e., the decomposition of the current snapshot tensor is *independent* of the previous snapshot tensor. Thanks to the CP decomposition of previous snapshot, DisMASTD could focus on the computation of relative complement of two consecutive tensor snapshots. In contrast, DMS-MG, as a traditional static tensor decomposition method, has to re-compute the tensor decomposition from the sketch, not able to benefit from the decomposition of previous tensors.

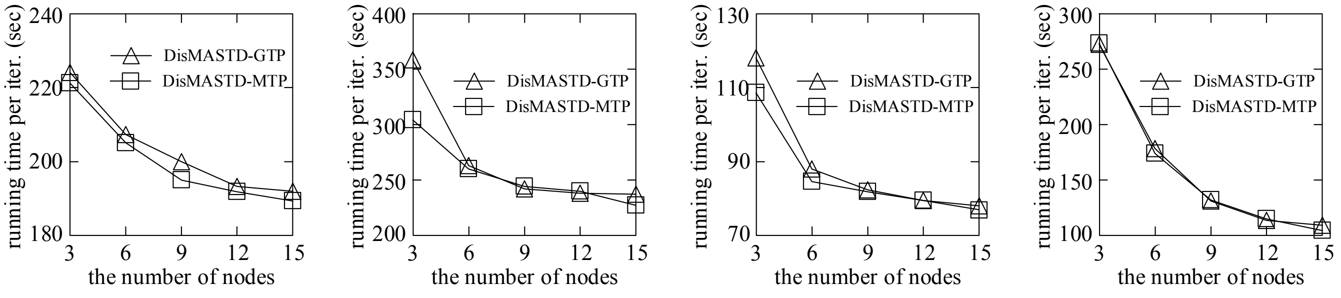
Third, as the tensor size grows, the running time of two DMS-MG methods increases while that of two DisMASTD methods stays much more stable. This clearly demonstrates the resilience of DisMASTD to the tensor size, which is very desirable in the era of big data. The reason is that the cost of DMS-MG depends on the number of non-zero elements within the tensor. However, the cost of DisMASTD depends on the number of non-zero elements in the relative complement of two consecutive tensor snapshots as discussed in Section IV-C. Besides, it is observed that the partitioning algorithm MTP performs slightly better than GTP. This is because MTP could achieve better load-balancing tensor partitioning than GTP. We will detail this in the next subsection.

2) Effect of the Tensor Partitioning: The second set of experiments is to compare the efficiency of the two different versions of our proposed tensor partitioning method and to explore the effect of the number of partitions.



(a) *Clothing* (b) *Book* (c) *Netflix* (d) *Synthetic*

Fig. 6. Running time per iteration versus the number of partitions



(a) *Clothing* (b) *Book* (c) *Netflix* (d) *Synthetic*

Fig. 7. Running time per iteration versus the number of nodes

Table IV lists the standard deviation statistics of *nnz* in the tensor partitioning result w.r.t. various the numbers of tensor partitions from 8 to 38 per mode, which could clearly demonstrate the load balancing of tensor partitioning. It is observed that the standard deviation values of MTP are smaller than that of GTP in all the real datasets (i.e., *Clothing*, *Book*, and *Netflix*). As for *Synthetic* dataset, the standard deviation values of both tensor partitioning methods are close. The reason behind is that the three real datasets are tensors with skewed non-zero element distribution while *Synthetic* dataset is randomly generated with uniform non-zero element distribution. This implies that MTP is more adaptive to the tensor with skewed non-zero element distribution as it partitions the tensor in over-all load balancing pattern. However, GTP simply assigns the partitioning boundaries as it greedily moves forward.

In addition, Fig. 6 illustrates the results under various numbers of tensor partitions from 8 to 38 per mode. The first observation is that the running time per iteration of both two DisMASTD methods first drops and then ascends or stays stable as the number of tensor partitions grows. This is because, the larger the number of partitions is, the more parallelism the distributed system has. However, more partitions require more overhead in the distributed system. Considering the trade-off above, it seems to be a good empirical setting to set the number of partitions to be the number of nodes in the distributed system. It is also observed that MTP performs slightly better than GTP.

3) Effect of the Number of Nodes: The third set of experiments is to verify the scalability of our proposed DisMASTD. Fig. 7 depicts the running time per iteration w.r.t. the number of nodes in the distributed system, which is changed from 3 nodes to 15 nodes. As expected, the running time drops as the number of nodes increases, since the computational resources grow with more nodes. It is worth mentioning that the speedup over the increasing number of nodes in *Synthetic* dataset is larger than that in the datasets with small nnz (e.g., *Clothing*). The reason is that the startup costs of Spark tasks dominate the running time when the datasets are small.

VI. Conclusion

In this paper, we study the problem of distributed multi-aspect streaming tensor decomposition. This is, to our best knowledge, the first attempt to tackle the problem in the distributed platform to support the arising high volume streaming data analysis. We propose DisMASTD, an efficient distributed multi-aspect streaming tensor decomposition. First, we prove the optimal partitioning problem is NP-hard. Second, we utilize two heuristic tensor partitioning methods, namely GTP and MTP, to achieve the load balancing. The former follows the order of slices, and tries to locate the boundary of each partition such that the number of non-zero elements within each partition is close to the target size. The latter allows each partition to contain non-adjacent slices, and strives to make sure the number of non-zero elements with each partition is close to each other. Third, we design a distributed multi-aspect streaming tensor decomposition computation method, which avoids repetitive computation and reduces the network communication by maintaining and reusing the intermediate results. Finally, we conduct extensive experiments using both real and synthetic datasets to confirm that DisMASTD is more efficient and meanwhile scales better than the state-of-the-art competitor.

Acknowledgments

This work was supported in part by the National Key R&D Program of China under Grant No. 2018YFB1004003, the NSFC under Grant No. 61972338, the NSFC Zhejiang Joint Fund under Grant No. U1609217, and the National Research Foundation, Prime Minister's Office, Singapore under its International Research Centres in Singapore Funding Initiative. Yunjun Gao is the corresponding author of the work.

References

[1] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.

[2] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: Models, applications, and scalable algorithms," *ACM TIST*, vol. 8, no. 2, pp. 16:1–16:44, 2017.

[3] Q. Song, H. Ge, J. Caverlee, and X. Hu, "Tensor completion algorithms in big data analytics," *TKDD*, vol. 13, no. 1, pp. 6:1–6:48, 2019.

[4] G. Vargas-Solar, J. Zechinelli-Martini, and J. Espinosa-Oviedo, "Big data management: What to keep from the past to face future challenges?" *Data Science and Engineering*, vol. 2, no. 4, pp. 328–345, 2017.

[5] M. T. Bahadori, Q. R. Yu, and Y. Liu, "Fast multivariate spatio-temporal analysis via low rank tensor learning," in *NIPS*, 2014, pp. 3491–3499.

[6] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal link prediction using matrix and tensor factorizations," *TKDD*, vol. 5, no. 2, pp. 10:1–10:27, 2011.

[7] Y. Wang, Y. Zheng, and Y. Xue, "Travel time estimation of a path using sparse trajectories," in *SIGKDD*, 2014, pp. 25–34.

[8] M. Signoretto, D. Q. Tran, L. D. Lathauwer, and J. A. K. Suykens, "Learning with tensors: A framework based on convex optimization and spectral regularization," *Machine Learning*, vol. 94, no. 3, pp. 303–351, 2014.

[9] H. Wang and N. Ahuja, "Facial expression decomposition," in *ICCV*, 2003, pp. 958–965.

[10] T. Yokota, B. Erem, S. Guler, S. K. Warfield, and H. Hontani, "Missing slice recovery for tensors using a low-rank model in embedded space," in *CVPR*, 2018, pp. 8251–8259.

[11] H. Fanaee-T and J. Gama, "Multi-aspect-streaming tensor analysis," *Knowl.-Based Syst.*, vol. 89, pp. 332–345, 2015.

[12] H. Ge, K. Zhang, M. Alfifi, X. Hu, and J. Caverlee, "DisTenC: A distributed algorithm for scalable tensor completion on spark," in *ICDE*, 2018, pp. 137–148.

[13] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," in *SIGKDD*, 2012, pp. 316–324.

[14] S. Oh, N. Park, L. Sael, and U. Kang, "Scalable tucker factorization for sparse tensors - algorithms and discoveries," in *ICDE*, 2018, pp. 1120–1131.

[15] K. Shin, L. Sael, and U. Kang, "Fully scalable methods for distributed tensor factorization," *IEEE Trans. Knowl. Data Eng.*, vol. 29, no. 1, pp. 100–113, 2017.

[16] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *IPDPS*, 2016, pp. 902–911.

[17] M. Mardani, G. Mateos, and G. B. Giannakis, "Subspace learning and imputation for streaming big data matrices and tensors," *IEEE Trans. Signal Processing*, vol. 63, no. 10, pp. 2663–2677, 2015.

[18] S. Zhou, X. V. Nguyen, J. Bailey, Y. Jia, and I. Davidson, "Accelerating online CP decompositions for higher order tensors," in *SIGKDD*, 2016, pp. 1375–1384.

[19] M. Nimishakavi, B. Mishra, M. Gupta, and P. P. Talukdar, "Inductive framework for multi-aspect streaming tensor completion with side information," in *CIKM*, 2018, pp. 307–316.

[20] Q. Song, X. Huang, H. Ge, J. Caverlee, and X. Hu, "Multi-aspect streaming tensor completion," in *SIGKDD*, 2017, pp. 435–443.

[21] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Journal of Mathematics and Physics*, vol. 6, no. 1-4, pp. 164–189, 1927.

[22] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.

[23] R. A. Harshman *et al.*, "Foundations of the parafac procedure: Models and conditions for an "explanatory" multimodal factor analysis," 1970.

[24] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.

[25] P. M. Kroonenberg and J. De Leeuw, "Principal component analysis of three-mode data by means of alternating least squares algorithms," *Psychometrika*, vol. 45, no. 1, pp. 69–97, 1980.

[26] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Analysis Applications*, vol. 21, no. 4, pp. 1253–1278, 2000.

[27] B. W. Bader and T. G. Kolda, "Efficient MATLAB computations with sparse and factored tensors," *SIAM J. Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2007.

[28] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *NIPS*, 2014, pp. 1296–1304.

[29] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing, "Flexifact: Scalable flexible factorization of coupled tensors on Hadoop," in *SDM*, 2014, pp. 109–117.

[30] E. E. Papalexakis, C. Faloutsos, T. M. Mitchell, P. P. Talukdar, N. D. Sidiropoulos, and B. Murphy, "Turbo-smt: Accelerating coupled sparse matrix-tensor factorizations by 200x," in *SDM*, 2014, pp. 118–126.

[31] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the CP format," *Parallel Computing*, vol. 57, pp. 222–234, 2016.

[32] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *ICDM*, 2014, pp. 989–994.

[33] B. Jeon, I. Jeon, L. Sael, and U. Kang, "Scout: Scalable coupled matrix-tensor factorization - algorithm and discoveries," in *ICDE*, 2016, pp. 811–822.

[34] N. Park, B. Jeon, J. Lee, and U. Kang, "BIGtensor: Mining billion-scale tensor made easy," in *CIKM*, 2016, pp. 2457–2460.

[35] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "SPLATT: Efficient and parallel sparse tensor-matrix multiplication," in *IPDPS*, 2015, pp. 61–70.

[36] S. Acer, T. Torun, and C. Aykanat, "Improving medium-grain partitioning for scalable sparse tensor decomposition," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 12, pp. 2814–2825, 2018.

[37] N. Park, S. Oh, and U. Kang, "Fast and scalable distributed boolean tensor factorization," in *ICDE*, 2017, pp. 1071–1082.

[38] ——, "Fast and scalable method for distributed boolean tensor factorization," *VLDB J.*, vol. 28, no. 4, pp. 549–574, 2019.

[39] D. Nion and N. D. Sidiropoulos, "Adaptive algorithms to track the PARAFAC decomposition of a third-order tensor," *IEEE Trans. Signal Processing*, vol. 57, no. 6, pp. 2299–2310, 2009.

[40] A. H. Phan and A. Cichocki, "PARAFAC algorithms for large-scale problems," *Neurocomputing*, vol. 74, no. 11, pp. 1970–1984, 2011.

[41] J. Sun, D. Tao, and C. Faloutsos, "Beyond streams and graphs: Dynamic tensor analysis," in *SIGKDD*, 2006, pp. 374–383.

[42] J. Sun, D. Tao, S. Papadimitriou, P. S. Yu, and C. Faloutsos, "Incremental tensor analysis: Theory and applications," *TKDD*, vol. 2, no. 3, pp. 11:1–11:37, 2008.

[43] R. Yu, D. Cheng, and Y. Liu, "Accelerated online low rank tensor learning for multivariate spatiotemporal streams," in *ICML*, 2015, pp. 238–247.

[44] W. Hu, X. Li, X. Zhang, X. Shi, S. J. Maybank, and Z. Zhang, "Incremental tensor subspace learning and its applications to foreground segmentation and tracking," *International Journal of Computer Vision*, vol. 91, no. 3, pp. 303–327, 2011.

[45] A. Sobral, C. G. Baker, T. Bouwmans, and E. Zahzah, "Incremental and multi-feature tensor subspace learning applied for background modeling and subtraction," in *ICIAR*, 2014, pp. 94–103.

[46] X. Ma, D. Schonfeld, and A. A. Khokhar, "Dynamic updating and downdating matrix SVD and tensor HOSVD for adaptive indexing and retrieval of motion trajectories," in *ICASSP*, 2009, pp. 1129–1132.

[47] R. E. Korf, "A complete anytime algorithm for number partitioning," *Artificial Intelligence*, vol. 106, no. 2, pp. 181–203, 1998.