2018 IEEE 34th International Conference on Data Engineering

DisTenC: A Distributed Algorithm for Scalable Tensor Completion on Spark

Hancheng Ge [*], Kai Zhang [†], Majid Alfifi [*], Xia Hu [*], James Caverlee [*]

[*] *Department of Computer Science and Engineering, Texas A&M University, USA*
[*] {hge,alfifi,hu,caverlee}@cse.tamu.edu

[†] *Computer&Information Sciences, Temple University, USA*
[†] zhang.kai@temple.edu

***Abstract*—How can we efficiently recover missing values for very large-scale real-world datasets that are multi-dimensional even when the auxiliary information is regularized at certain mode? Tensor completion is a useful tool to recover a low-rank tensor that best approximates partially observed data and further predicts the unobserved data by this low-rank tensor, which has been successfully used for many applications such as location-based recommender systems, link prediction, targeted advertising, social media search, and event detection. Due to the curse of dimensionality, existing algorithms for tensor completion that integrate auxiliary information do not scale for tensors with billions of elements. In this paper, we propose DISTENC, a new distributed large-scale tensor completion algorithm that can be distributed on Spark. Our key insights are to (i) efficiently handle trace-based regularization terms; (ii) update factor matrices with caching; and (iii) optimize the update of the new tensor via residuals. In this way, we can tackle the high computational costs of traditional approaches and minimize intermediate data, leading to order-of-magnitude improvements in tensor completion. Experimental results demonstrate that DISTENC is capable of handling up to $10 \sim 1000\times$ larger tensors than existing methods with much faster convergence rate, shows better linearity on machine scalability, and achieves up to an average improvement of 23.5% in accuracy in applications.**

I. INTRODUCTION

Extremely large and sparse multi-dimensional data arise in a number of important applications, including location-based recommendation, targeted advertising, social media search, and event detection [1], [2], [3]. Tensors – or multi-dimensional arrays – are commonly used to capture this multi-dimensionality. For instance, a movie rating from a user can be modeled as a tensor where each element is an interaction between a movie, a user, and the context in which this user rates the movie (e.g., genre, date of the rating, etc.). A multi-dimensional social network such as the DBLP network can be represented as a tensor with 4-tuples, e.g., *author-paper-term-venue*. Analytics over such large, diverse, and multi-dimensional datasets can provide valuable insights with respect to the underlying relationships between different entities.

However, in practice, many types of multidimensional data may be *noisy* or *incomplete*, limiting the effectiveness of such analytics. For example, data may be restricted due to data sampling policies, partial access to legacy data warehouses, sparse feedback from users (e.g., ratings in a recommender system), data missing at random, and so on [4], [5]. Traditional methods like matrix completion methods have shown good success in recovering two-dimensional data, but may not be suitable for handling missing data in these large multi-dimensional cases. Analogous to matrix completion, *tensor completion* aims to recover a low-rank tensor that best approximates partially observed data and further predicts the unobserved data using this low-rank tensor.

While recovering the missing values by tensor completion is attractive, it is challenging to efficiently handle large-scale tensors (e.g., ones containing billions of observations in each mode) due to the high computational costs and space requirements. Tensor completion in these scenarios faces challenges such as: (i) the intermediate data explosion problem where in updating factor matrices, the amount of intermediate data of an operation exceeds the capacity of a single machine or even a cluster [6], [7], [8], [9]; (ii) the large regularization problem where the regularization term can affect the scalability and parallelism of tensor completion [10], [11]; and (iii) since architectures on modern computing facilities have lower ratios of memory bandwidth to compute capabilities, computations on tensors that usually have unstructured access patterns are usually degraded. While there has been research addressing these challenges of scalability separately, most focus on tensor factorization, which are not suitable for tensor completion that needs to estimate all missing values in a tensor at each iteration. There is a need to fill a gap between tensor completion and applications with real large-scale datasets.

In this paper, we propose to fill this gap through DISTENC (Distributed Tensor Completion), a new distributed large-scale tensor completion algorithm running on Apache Spark. Our intuition is to tackle the challenges of large-scale tensor completion through three key insights: (i) by designing an efficient algorithm for handling the trace-based regularization term; (ii) by updating factor matrices with caching; and (iii) by optimizing the update of the new tensor at each iteration, while minimizing the generation and shuffling of intermediate data. We find that DISTENC leads to high efficiency compared with state-of-the-art methods, while delivering similar (and in many cases improved) accuracy. The three main contributions of this paper are as follows:

- **Algorithm.** We propose DISTENC, a novel distributed tensor completion algorithm with regularized trace of the information based on ADMM, which is designed to scale up to real large-scale tensors by efficiently computing auxiliary variables, minimizing intermediate data, and reducing the workload of updating new tensors.

- **Scalability** Our scalability analysis of DISTENC shows that it achieves up to $10 \sim 1000\times$ better scalability, performs better linearity as we scale the number of machines, and converges faster than other methods. Additionally, we analyze DISTENC in terms of time complexity, memory requirement and the amount of shuffled data.
- **Experiment.** We empirically evaluate DISTENC and confirm its superior scalability and performance in tensor completion with both synthetic and real-world datasets. We observe that the proposed DISTENC performs tensor completion with less time and up to an average improvement of 23.5% in accuracy in applications versus state-of-the-art methods, while achieving better scalability.

II. PRELIMINARIES

In this section, we provide a brief background on tensors including key definitions and notations, followed by the tensor completion. Table I lists the symbols used in this paper.

A. Tensor

Definition 2.1.1 (Tensor). A tensor is a multi-way array, whose dimension is called *mode* or *order*. An Nth-order tensor is an N-mode array, denoted as $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$. The number of non-zeros of a tensor $\mathcal{X}$ is denoted as $nnz(\mathcal{X})$.

Definition 2.1.2 (Kronecker Product). Given two matrices $\mathbf{A} \in \mathbb{R}^{I\times J}$ and $\mathbf{B} \in \mathbb{R}^{K\times L}$, their Kronecker product $\mathbf{A} \otimes \mathbf{B}$ generates a matrix of size $IK \times JL$, which can be defined as:

$$\mathbf{A} \otimes \mathbf{B} = \begin{bmatrix} a_{11}\mathbf{B} & a_{11}\mathbf{B} & \cdots & a_{1J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}\mathbf{B} & a_{11}\mathbf{B} & \cdots & a_{IJ}\mathbf{B} \end{bmatrix}.$$

Definition 2.1.3 (Khatri-Rao Product). It is a column-wise Kronecker product, denoted as $\mathbf{A} \odot \mathbf{B}$ where both $\mathbf{A} \in \mathbb{R}^{I\times R}$ and $\mathbf{B} \in \mathbb{R}^{K\times R}$ have the same number of columns. Their Khatri-Rao product produces a matrix of size $IK \times R$ defined:

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1, \cdots, \mathbf{a}_R \otimes \mathbf{b}_R].$$

Definition 2.1.4 (Hadamard Product). Given two matrices $\mathbf{A}$ and $\mathbf{B}$ with the same size $I \times J$, their Hadamard product $\mathbf{A} \times \mathbf{B}$ is the element-wise matrix product, defined as:

$$\mathbf{A} \times \mathbf{B} = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{bmatrix}. \quad (1)$$

Definition 2.1.5 (Tensor Matricization). Tensor matricization is to unfold a tensor into a matrix format with a predefined sequence of mode order. The *n*-mode matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ is denoted as $\mathcal{X}_{(n)} \in \mathbb{R}^{I_n \times (\prod_{k\neq n} I_k)}$. The order of the other modes except mode n can be arranged randomly to construct the column of $\mathcal{X}_{(n)}$.

TABLE I: Symbols and Operations.

Symbols	Definitions
$\mathcal{X}$	tensor (Euler script letter)
$\mathbf{X}_{(n)}$	n-mode matricization of a tensor $\mathcal{X}$
$\mathbf{X}$	matrix (uppercase bold letter)
$\mathbf{x}$	column vector (lowercase bold letter)
x	scalar (lowercase letter)
N	order of a tensor (number of modes)
$[\![\cdot]\!]$	Kruskal operator
$\otimes$	Kronecker product
$\odot$	Khatri-Rao product
$*$	Hadamard product
$\circ$	outer product
$\times_n$	n-mode tensor-matrix product
$\|\mathbf{X}\|_F^2$	Frobenius norm of $\mathbf{X}$
$nnz(\mathcal{X})$	number of non-zero elements in $\mathcal{X}$

Definition 2.1.5 (n-mode Tensor-Matrix Product). Given an N^{th}-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_N}$ and a matrix $\mathbf{A} \in \mathbb{R}^{I_n \times J}$, their multiplication on its n^{th}-mode is represented as $\mathcal{Y} = \mathcal{X} \times_n \mathbf{A}$ and is of size $I_1 \times \ldots I_{n-1} \times J \times I_{n+1} \ldots \times I_N$. The element-wise result is illustrated as:

$$\mathcal{Y}_{i_1,\ldots,i_{n-1},j,i_{n+1},\ldots,i_N} = \sum_{k=1}^{I_n} \mathcal{X}_{i_1,\ldots,i_{n-1},k,i_{n+1},\ldots,i_N} \mathbf{A}_{k,j}. \quad (2)$$

B. Tensor Completion

Tensor completion is extensively applied in tensor mining to fill the missing elements with partially observed tensors. *Low rank* is often a necessary hypothesis to restrict the degree of freedoms of the missing entries being intractable. Hence, we focus on the low-rank tensor completion (LRTC) problem in this paper. First, we introduce the standard CANDECOMP/PARAFAC(CP)-based tensor completion.

1) CP-based Tensor Completion: CP tensor decomposition proposed by Hitchcock [12] is one of the most used tensor factorization models, which decomposes a tensor into a sum of rank-one tensors. Before being actively researched in recent years, the LRTC problem is usually considered as a byproduct of the tensor decomposition problem with missing values. Given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with the rank $R \ll min(I_1, \ldots, I_N)$ that is pre-defined as one of the inputs, the CP decomposition solves:

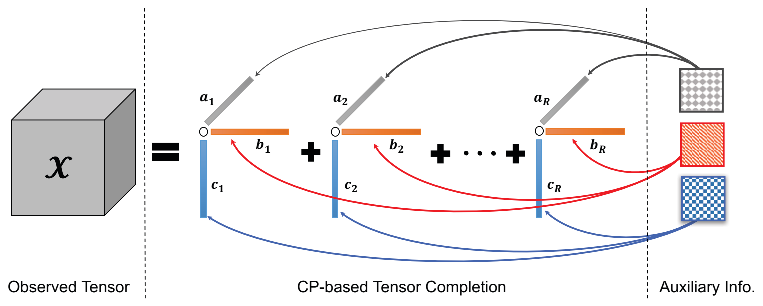
$$\begin{aligned} &\underset{\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)},\mathcal{X}}{\text{minimize}} && \frac{1}{2}\|\mathcal{X} - [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]\|_F^2 + \frac{\lambda}{2}\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 \\ &\text{subject to} && \mathbf{\Omega} * \mathcal{X} = \mathcal{T}, \mathbf{A}^{(n)} \geq 0, n = 1, 2, 3., \end{aligned}$$

where $\mathcal{T}$ denotes the partial observations, $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times R}$ are the factor matrices, and $\mathbf{\Omega}$ is a non-negative weight tensor with the same size as $\mathcal{X}$:

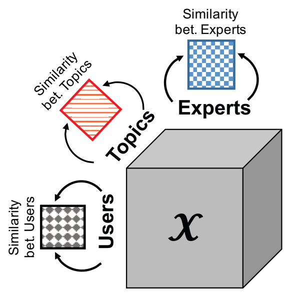
$$\mathbf{\Omega}_{i_1\ldots i_n\ldots i_N} = \begin{cases} 1 & \text{if } \mathcal{X}_{i_1\ldots i_n\ldots i_N} \text{ is observed,} \\ 0 & \text{if } \mathcal{X}_{i_1\ldots i_n\ldots i_N} \text{ is unobserved.} \end{cases}$$

Through the CP decomposition, an N^{th}-order tensor $\mathcal{X}$ is decomposed into N factor matrices as follows:

$$\mathcal{X} \approx [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!] = \sum_{i=1}^{R} \mathbf{a}_i^{(1)} \circ \mathbf{a}_i^{(2)} \circ \ldots \circ \mathbf{a}_i^{(N)}, \quad (3)$$



(a) Tensor Completion with Auxiliary Information



(b) Twitter List

Fig. 1: Example of CP-based tensor completion with auxiliary information. (a): Rank-R CP tensor completion of a 3-order tensor with auxiliary information. The tensor $\mathcal{X}$ is decomposed into three factor matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$, and recovered based on factor matrices. (b) A 3-order Twitter List tensor with *user-expert-topic* triples and three similarity matrices generated from auxiliary information of users, experts and topics, respectively.

where $\mathbf{a}_i^{(n)}$ is the i-th column of matrix $\mathbf{A}_n$.

2) Tensor Completion with Auxiliary Information: With the increasing ratio of missing entries, tensor completion may perform unsatisfactory imputation with degrading accuracy due to its assumptions on low-rank and uniformly sampling. In real-world data-driven applications, besides the target tensor object, considerable additional auxiliary information such as spatial and temporal similarities among objects or auxiliary coupled matrices/tensors may also exist and provide potential help for improving completion quality. An example of a Twitter List tensor is illustrated in Fig. 1b. Given an Nth-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with the rank $R \ll min(I_1, \ldots, I_N)$ and similarity matrices $\mathbf{B}^{(n)}, n = 1, \ldots, N$ of size $I_1 \times I_1, \ldots, I_N \times I_N$, the tensor decomposition with auxiliary information solves:

$$\begin{aligned} \underset{\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)},\mathcal{X}}{\text{minimize}} \quad & \frac{1}{2}\|\mathcal{X} - [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]\|_F^2 \\ & + \frac{\lambda}{2}\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 + \sum_{n=1}^{N}\alpha_n \text{tr}(\mathbf{B}^{(n)^T}\mathcal{L}_n\mathbf{B}^{(n)}) \\ \text{subject to} \quad & \mathbf{\Omega} * \mathcal{X} = \mathcal{T}, \mathbf{A}^{(n)} = \mathbf{B}^{(n)} \geq 0, n = 1, 2, \ldots, N., \end{aligned} \tag{4}$$

where $\mathcal{L}_n \in \mathbb{R}^{I_n \times I_n}$ is the graph Laplacian of the similarity matrix $\mathbf{S}_n$ for the mode n, $\mathbf{B}^{(n)}, n = 1, 2, \ldots, N$ are introduced as auxiliary variables, $tr(\cdot)$ is the matrix trace and α_n is to control the weight of auxiliary information in the mode n. Fig. 1a shows the rank-R CP tensor completion of a 3-order tensor with auxiliary information. The tensor $\mathcal{X}$ is decomposed into three factor matrices $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ by integrating auxiliary information and recovered based on them.

3) Optimization Algorithm: Since the objective function in Eq.(4) is convex with respect to variables $\mathbf{A}^{(n)}$ and $\mathbf{B}^{(n)}$ separately, the overall problem is not convex. Motivated by methods [13], [14], we can construct an algorithm under the framework of ADMM (Alternating Direction Method of Multipliers) to find optimal solutions for the objective function above. ADMM [15] has illustrated its superiority over alternating least square (ALS) in terms of both reconstruction efficiency and accuracy [16]. In order to apply ADMM, the objective function in Eq. (4) can be firstly written in the partial augmented Lagrangian functions as follow:

$$\begin{aligned} L_\eta(\mathbf{A}^{(n)}, \mathbf{B}^{(n)}, \mathbf{Y}^{(n)}) &= \frac{1}{2}\|\mathcal{X} - [\![\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \ldots, \mathbf{A}^{(N)}]\!]\|_F^2 \\ &+ \frac{\lambda}{2}\sum_{n=1}^{N}\|\mathbf{A}^{(n)}\|_F^2 + \sum_{n=1}^{N}\frac{\alpha_n}{2}\text{tr}(\mathbf{B}^{(n)^T}\mathcal{L}_n\mathbf{B}^{(n)}) \\ &+ \sum_{n=1}^{N} < \mathbf{Y}^{(n)}, \mathbf{B}^{(n)} - \mathbf{A}^{(n)} > + \sum_{i=1}^{N}\frac{\eta}{2}\|\mathbf{B}^{(n)} - \mathbf{A}^{(n)}\|_F^2, \end{aligned} \tag{5}$$

where $\mathbf{Y}^{(n)}$ is the matrix of Lagrange multipliers for $n = 1, 2, \ldots, N$, η is a penalty parameter and $< *, * >$ is an inner product of matrices. The variables $\mathbf{A}^{(n)}, \mathbf{B}^{(n)}, \mathbf{Y}^{(n)}, n = 1, 2, \ldots, N$ can be iteratively updated by calculating the partial derivatives while fixing other variables, as shown in Algorithm 1 detailed in [4]. There are many ways to check for convergence. The stopping criterion for Algorithm 1 is either one of the following: (i) the maximal difference between factor matrices of consecutive iterations is smaller than a threshold; or (ii) the maximum number of iterations is exceeded.

III. Proposed Method: DisTenC

In this section, we present the proposed DisTenC, a distributed algorithm for scalable tensor completion on Spark.

A. Overview

DisTenC provides an efficient distributed algorithm for the CP-based tensor completion with auxiliary information on Spark. The algorithm 1 contains three challenging operations: (i) updating auxiliary variables $\mathbf{B}^{(n)}$ (line 4); (ii) updating factor matrices $\mathbf{A}^{(n)}$ (lines 7 and 8); and (iii) updating tensor $\mathcal{X}$ (line 9). In the following subsections, we address the above challenges with the following main ideas that efficiently update auxiliary variables, factor matrices and tensors in distributed systems, while reducing floating point operations (FLOPs).

- (Section III-B) Eigen-decomposing a graph Laplacian matrix and carefully ordering of computation to decrease FLOPs in updating auxiliary variables.

Algorithm 1: CP-based Tensor Completion via ADMM

```
   Input: $\mathcal{T}, \mathbf{A}_0^{(n)}, \boldsymbol{\Omega}, \boldsymbol{\Omega}^c, \lambda, \rho, \eta, \eta_{max}, N$
   Output: $\mathcal{X}, \mathbf{A}^{(n)}, \mathbf{B}^{(n)}, \mathbf{Y}^{(n)}$
1  Initialize $\mathbf{A}_0^{(n)} \geq 0$, $\mathbf{B}_0^{(n)} = \mathbf{Y}_0^{(n)} = 0$, $t = 0$
2  while Not Converged do
3      for $n \leftarrow 1$ to $N$ do
4          Update
             $\mathbf{B}_{t+1}^{(i)} \leftarrow (\eta_t \boldsymbol{I} + \alpha_n \boldsymbol{\mathcal{L}}_n)^{-1}(\eta_t \mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)})$
5          Calculate $\mathbf{U}_t^{(n)} \leftarrow$
6          $(\mathbf{A}_t^{(N)} \odot \cdots \odot \mathbf{A}_t^{(n+1)} \odot \mathbf{A}_t^{(n-1)} \odot \cdots \odot \mathbf{A}_t^{(1)})$
7          Update $\mathbf{A}_{t+1}^{(n)} \leftarrow$
8          $(\mathbf{X}_{(n)}^t \mathbf{U}_t^{(n)} + \eta_t \mathbf{B}_{t+1}^{(n)} + \mathbf{Y}_t^{(n)})(\mathbf{U}_t^{(n)^T} \mathbf{U}^{(n)_t} +$
             $\lambda \boldsymbol{I} + \eta_t \boldsymbol{I})^{-1}$
9      Update $\mathcal{X}^{t+1} = \mathcal{T} + \boldsymbol{\Omega}^c * [\![\mathbf{A}_{t+1}^{(1)}, \mathbf{A}_{t+1}^{(2)}, \ldots, \mathbf{A}_{t+1}^{(N)}]\!]$
10     for $n \leftarrow 1$ to $N$ do
11         Update $\mathbf{Y}_{t+1}^{(n)} = \mathbf{Y}_t^{(n)} + \eta_t(\mathbf{B}_{t+1}^{(n)} - \mathbf{A}_{t+1}^{(n)})$
12     Update $\eta_{t+1}$ =min$(\rho\eta_t, \eta_{max})$
13     Check the convergence:
         $\max\{\|\mathbf{A}_{t+1}^{(n)} - \mathbf{B}_{t+1}^{(n)}\|_F, n = 1, 2, \ldots, N\} < tol$
14     $t = t + 1$
15 return $\mathcal{X}$, $\mathbf{A}^{(n)}, n = 1, 2, \ldots, N$
```

- (Section III-C) Carefully partitioning of the workload and distributing intermediate generation to remove redundant data generation and reducing the amount of intermediate data transfer in updating factor matrices.
- (Section III-D) Utilizing the residual tensor to avoid the explicit computation of the dense tensor and reuse intermediate data to decrease FLOPs in updating tensor.

B. Calculating Inverse of Graph Laplacian Matrices

Since the update rules for auxiliary variables $\mathbf{B}^{(n)}, n = 1, 2, \ldots, N$ are similar, we focus on updating the variable $\mathbf{B}^{(n)}$ where n could be an arbitrary one from $\{1, 2, \ldots, N\}$. The operation in line 4 of Algorithm 1 requires us to compute the pseudo-inverse of the summation of a matrix $\alpha_n \boldsymbol{\mathcal{L}}_n$ and a diagonal matrix $\eta_t \boldsymbol{I}$ where $\boldsymbol{I}$ is an identity matrix with the same size of $\boldsymbol{\mathcal{L}}_n$. Since such summation will change with the penalty parameter η_t that will be updated at every iteration, the question is how to efficiently calculate such a pseudo-inverse instead of computing it at every iteration due to its high computational cost with complexity $\mathcal{O}(I_n^3)$.

As a graph Laplacian matrix $\boldsymbol{\mathcal{L}}_n$ derived from the similarity matrix $\mathbf{S}_n$ is symmetric and predefined without any change in Algorithm 1, we apply an efficient truncated eigen-decomposition method proposed by Bientinest et al. [17] with the time complexity $\mathcal{O}(KI_n)$ and the space complexity $\mathcal{O}(I_n)$ to it and get its truncated decomposition as $\boldsymbol{\mathcal{L}}_n = \mathbf{V}_n \boldsymbol{\Lambda}_n \mathbf{V}_n^T$ where $\mathbf{V}_n \in \mathbb{R}^{I_n \times K}$ and $\boldsymbol{\Lambda}_n \in \mathbb{R}^{K \times K}$. Hence, line 4 of Algorithm 1 can be re-written as follow:

$$\mathbf{B}_{t+1}^{(n)} \leftarrow \mathbf{V}_n(\eta_t + \alpha_n \Lambda_n)^{-1} \mathbf{V}_n^T (\eta_t \mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)}). \quad (6)$$

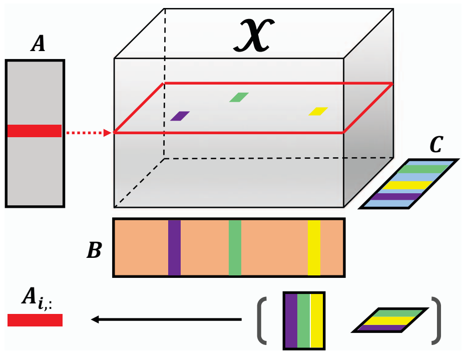


Fig. 2: Memory access on updating $\mathbf{A}_{i,:}$ in a 3-order tensor.

Since $(\eta_t + \alpha_n \boldsymbol{\Lambda}_n)$ is a diagonal matrix, its inverse can be efficiently computed by only computing the reciprocal of entries on the diagonal instead of calculating the inverse of the whole matrix $(\eta_t \boldsymbol{I} + \alpha_n \boldsymbol{\mathcal{L}}_n)$. Furthermore, Eq.(6) performs the matrix multiplication of the four matrices $\mathbf{V}_n$, $(\eta_t + \alpha_n \Lambda_n)^{-1}$, $\mathbf{V}_n^T$ and $(\eta_t \mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)})$. Its computing order may significantly affect the efficiency of computation. In order to reduce FLOPs, we compute it by firstly multiplying the last two matrices that result in a relatively small matrix with size $K \times R$, and then broadcasting the result with the second one to the first matrix:

$$\mathbf{B}_{t+1}^{(n)} \leftarrow \mathbf{V}_n(\eta_t + \alpha_n \Lambda_n)^{-1} (\mathbf{V}_n^T (\eta_t \mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)})). \quad (7)$$

By this way, we are able to perform the update of an auxiliary variable $\mathbf{B}^{(n)}$ in $\mathcal{O}(I_n R + I_n K R + I_n K^2 R)$ time.

C. Reducing Intermediate Data

As shown in lines 7 and 8 in Algorithm 1, we focus on the updating rule for an arbitrary $\mathbf{A}^{(n)}$ as follow:

$$\mathbf{A}^{(n)} \leftarrow (\mathbf{X}_{(n)} \mathbf{U}^{(n)} + \eta \mathbf{B}^{(n)} + \mathbf{Y}^{(n)})(\mathbf{U}^{(n)^T} \mathbf{U}^{(n)} + \lambda \boldsymbol{I} + \eta \boldsymbol{I})^{-1}. \quad (8)$$

where $\mathbf{U}^{(n)} = (\mathbf{A}^{(N)} \odot \cdots \odot \mathbf{A}^{(n+1)} \odot \mathbf{A}^{(n-1)} \odot \cdots \odot \mathbf{A}^{(1)})$ with size $(\prod_{k \neq n} I_k) \times R$, which entails three matrix-matrix multiplications as:

$$\begin{aligned} \boldsymbol{\mathcal{H}}_1 &= \mathbf{X}_{(n)} \mathbf{U}^{(n)}, \\ \boldsymbol{\mathcal{H}}_2 &= (\mathbf{U}^{(n)^T} \mathbf{U}^{(n)} + \lambda \boldsymbol{I} + \eta \boldsymbol{I})^{-1}, \\ \boldsymbol{\mathcal{H}}_3 &= (\boldsymbol{\mathcal{H}}_1 + \eta \mathbf{B}^{(n)} + \mathbf{Y}^{(n)}) \boldsymbol{\mathcal{H}}_2. \end{aligned} \quad (9)$$

We denote $\boldsymbol{\mathcal{H}}_1 = \mathbf{X}_{(n)} \mathbf{U}^{(n)}$ as the *matricized tensor times Khatri-Rao product* (MTTKRP) that will lead to the intermediate data explosion problem in the tensor completion when tensor $\mathcal{X}$ is very large. Explicitly calculating $\mathbf{U}^{(n)}$ and performing the matrix multiplication with $\mathbf{X}_{(n)}$ requires more memory than what a common cluster can afford as computing $\mathbf{U}^{(n)}$ is prohibitively expensive with the size $(\prod_{k \neq n} I_k) \times R$. Though the matricized $\mathbf{X}_{(n)}$ is very sparse, $\mathbf{U}^{(n)}$ is very large and dense. Hence, inspired by the work [8], we perform MTTKRP in place by exploiting the block structure of the Khatri-Rao product. A block is defined as a unit of workload distributed across machines, which determines the level of parallelism and the amount of shuffled data. For a better

Algorithm 2: DisTenC-Greedy Algorithm

```
Input: observed tensor $\mathcal{T}$, number of partitions at
       $n$-mode $P_n$, number of modes $N$
Output: $\boldsymbol{w}_n, n = 1, \cdots, N$
1  for $n = 1, \cdots, N$ do
2      $\delta = nnz(\mathcal{T})/P_n \leftarrow$ Calculate the chunk size for
       $n$-mode;
3      $sum = 0$ and $\epsilon_{pre} = \delta$;
4      $\boldsymbol{\theta}^{(n)} \leftarrow$ Calculate $nnz$ for each slice at $n$-mode;
5      for $i = 1, \cdots, I_n$ do
6          $sum \leftarrow sum + \boldsymbol{\theta}_i^{(n)}$;
7          $\epsilon \leftarrow$ Calculate the difference between $sum$ and $\delta$;
8          if $sum \geq \delta$ then
9              $\boldsymbol{w}_n \leftarrow$ add $i$ if $\epsilon < \epsilon_{pre}$; otherwise, add $i-1$;
10         $\epsilon_{pre} = \epsilon$;
```

illustration, we assume that $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is a 3-order sparse tensor whose entry $\mathcal{H}_1(i_1, r)$ can be represented as:

$$\mathcal{H}_1(i_1, r) = \sum_{\mathcal{X}_{i_1,:,:}} \mathcal{X}_{i_1,i_2,i_3} \mathbf{A}^{(3)}_{i_3,r} \mathbf{A}^{(2)}_{i_2,r} \tag{10}$$

As shown in Eq.(10), we observe two important properties of MTTKRP: (i) non-zeros in $\mathcal{X}_{i_1,:,:}$ are only associated with the computation of $\mathcal{H}_1(i_1, :)$; (ii) the row indices i_2 and i_3 in $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$, respectively, will be accessed based upon which appear in non-zeros in $\mathcal{X}_{i_1,:,:}$ when calculating $\mathcal{H}_1(i_n, r)$. Thus, our idea is to *compartmentalize* the sparse tensor $\mathcal{X}$ and factor matrices $\mathbf{A}^{(1)}, \mathbf{A}^{(2)}, \cdots, \mathbf{A}^{(N)}$ into blocks in order to make the computation of MTTKRP fit into the memory. Taking a 3-order tensor as an example, we divide rows of $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ into P, Q, and K blocks, respectively. Correspondingly, the tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ can be further divided into $P \times Q \times K$ blocks. A block of a tensor is denoted as $\mathcal{X}(p, q, k)$ with corresponding blocks of factor matrices $\mathbf{A}^{(1)}_{(p)}$, $\mathbf{A}^{(2)}_{(q)}$ and $\mathbf{A}^{(3)}_{(k)}$. Each process only work on a block of a factor matrix with entries in the tensor with which this block is associated, and aggregate partial results computed by other processors for this block.

Load Balancing. Since the tensor $\mathcal{X}$ is very sparse, randomly dividing it into $P \times Q \times K$ blocks could result in *load imbalance* [18]. The questions is how to identify the block boundaries. In order to fully utilize the computing resources, a greedy algorithm is proposed to generate blocks for balancing the workload. For instance, we split a mode into P partitions. Each partition will be generated by continuously adding indices until the number of non-zeros in this partition is equal to or larger than $nnz(\mathcal{X})/P$ that is considered as the target partition size. Once adding a slice makes a partition over the target size, we compare the number of non-zeros in this partition before and after adding it and pick whichever is closer to the target size. Other modes would follow the same routine to identify boundaries for Q partitions and K partitions. The algorithm for balancing the load for DisTenC is demonstrated in Algorithm 2, which will take $\mathcal{O}(Nnnz(\mathcal{X}))$.

Computing MTTKRP. After compartmentalizing the tensor and factor matrices into blocks, each process holds the tensor non-zeros with necessary blocks of factor matrices (non-local factor matrix rows will be transfered to this process from others), and performs MTTKRP as shown in Eq.(10). Specifically, we parallelize such computation based on the efficient fiber-based data structure [8] in local, indicating that we directly calculate the row of $\mathcal{H}_1$ as follow:

$$\mathcal{H}_1(i_1, :) = \sum_{\mathcal{X}_{i_1,:,:}} \mathcal{X}_{i_1,i_2,i_3} (\mathbf{A}^{(3)}_{i_3,:} * \mathbf{A}^{(2)}_{i_2,:}). \tag{11}$$

Since such calculation can be done at the granularity of factor matrix rows, it only requires $\mathcal{O}(R)$ intermediate memory per thread in parallelism. By this way, $\mathcal{H}_1$ are row-wise computed and distributed among all processes. We only need to broadcast relatively small factor matrices along with corresponding indices in the non-zeros of a sparse tensor for each machine instead of having to compute the entire Khatri-Rao product.

Calculating $\mathbf{U}^{(n)^T}\mathbf{U}^{(n)}$ Based upon the property of Khatri-Rao product, we can re-write $\mathbf{U}^{(1)^T}\mathbf{U}^{(1)}$ as follow:

$$\mathbf{U}^{(1)^T}\mathbf{U}^{(1)} = \mathbf{A}^{(3)^T}\mathbf{A}^{(3)} * \mathbf{A}^{(2)^T}\mathbf{A}^{(2)}. \tag{12}$$

By this way, we avoid explicitly computing the large intermediate matrix $\mathbf{U}^{(1)}$ with size $I_2 I_3 \times R$ by calculating the self-products $\mathbf{A}^{(2)^T}\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)^T}\mathbf{A}^{(3)}$ of factor matrices $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$. Applying the block matrix form, the computation of $\mathbf{A}^{(1)^T}\mathbf{A}^{(1)}$ can be represented in a distributed fashion as:

$$\mathbf{A}^{(1)^T}\mathbf{A}^{(1)} = \sum_{p=1}^{P} \mathbf{A}^{(1)^T}_{(p)} \mathbf{A}^{(1)}_{(p)}. \tag{13}$$

Each process calculates a local $\mathbf{A}^{(1)^T}_{(p)} \mathbf{A}^{(1)}_{(p)}$ in the thread-level parallelism. By aggregating all computations across processes, the final matrix $\mathbf{A}^{(1)^T}\mathbf{A}^{(1)}$ will be generated and distributed among all processes, which is a matrix of size $R \times R$ that can easily fit into the memory of each process. Thus, it can be seen that $(\mathcal{H}_2 + \lambda \boldsymbol{I} + \eta \boldsymbol{I})^{-1}$ can be efficiently calculated in $\mathcal{O}(R^3)$ time in a single machine.

D. Computing the Updated Tensor

Unlike the tensor factorization/decomposition in which the input tensor is fixed, tensor completion requires us to update the tensor $\mathcal{X}$ by filling out unobserved elements in each iteration as shown in line 9 in Algorithm 1. Once completing updates of $\mathbf{A}^{(1)}, \cdots, \mathbf{A}^{(N)}$ in an iteration, unobserved elements in a sparse tensor will be filled out by estimated values. Thus, it turns out to be a dense tensor that leads to a significant increase in the computation of updating factor matrices in lines 7 and 8 in Algorithm 1. The question is how to avoid such a problem and keep the computation in the level of $\mathcal{O}(nnz(\mathcal{X}))$ time. First of all, we define the residual tensor as:

$$\mathcal{E} = \boldsymbol{\Omega} * (\mathcal{T} - [\![\mathbf{A}^{(1)}, \ldots, \mathbf{A}^{(N)}]\!]), \tag{14}$$

which is sparse with the same size of the observed sparse tensor $\mathcal{T}$. Based upon the definition of *tensor matricizied*, its *n*-mode matricized form can be expressed as:

$$\mathcal{X}_{(n)} \approx \mathbf{A}^{(n)}(\mathbf{A}^{(N)}\odot\cdots\mathbf{A}^{(n+1)}\odot\mathbf{A}^{(n+1)}\odot\cdots\mathbf{A}^{(1)})^T. \quad (15)$$

For brevity, we take a 3-order tensor as an example as demonstrated in the previous section. By leveraging the residual tensor and the 1-mode matricized form of a tensor, we can re-write $\mathcal{H}_1^{t+1}$ shown in the Eq.(9) as:

$$\begin{aligned}\mathcal{H}_1^{t+1} &= \mathbf{X}_{(1)}^t\mathbf{U}_t^{(1)}\\ &= ([\![\mathbf{A}_t^{(1)},\mathbf{A}_t^{(2)},\mathbf{A}_t^{(3)}]\!]_{(1)} + \mathcal{E}_{(1)}^t)\mathbf{U}_t^{(1)}\\ &= [\![\mathbf{A}_t^{(1)},\mathbf{A}_t^{(2)},\mathbf{A}_t^{(3)}]\!]_{(1)}\mathbf{U}_t^{(1)} + \mathcal{E}_{(1)}^t\mathbf{U}_t^{(1)}\\ &= \mathbf{A}_t^{(1)}(\mathbf{A}_t^{(3)}\odot\mathbf{A}_t^{(2)})^T(\mathbf{A}_t^{(3)}\odot\mathbf{A}_t^{(2)}) + \mathcal{E}_{(1)}^t\mathbf{U}_t^{(1)}\\ &= \mathbf{A}_t^{(1)}(\mathbf{U}_t^{(1)T}\mathbf{U}_t^{(1)}) + \mathcal{E}_{(1)}^t\mathbf{U}_t^{(1)}.\end{aligned} \quad (16)$$

We see that $\mathcal{H}_1^{t+1}$ consists of two parts that are able to reduce the time complexity to $\mathcal{O}(nnz(\mathcal{X}))$. Concretely, the first part $\mathbf{A}_t^{(1)}(\mathbf{U}_t^{(1)T}\mathbf{U}_t^{(1)})$ takes $\mathcal{O}((I_1+I_2+I_3)R^2)$ FLOPs as shown in the computation of $\mathcal{H}_2$ of the section III-C; the second part $\mathcal{E}_{(1)}^t\mathbf{U}_t^{(1)}$ is performed by the method illustrated in the section III-C with the complexity $\mathcal{O}(nnz(\mathcal{X}))$ since it is only related to the residual tensor $\mathcal{E}$ instead of using a updated dense tensor. Fortunately, each $\mathbf{U}_t^{(1)T}\mathbf{U}_t^{(1)}$ is computed during the iteration and the results can be cached and reused in only $\mathcal{O}(R^2)$ space.

E. Complexity Analysis

We now analyze the proposed DISTENC algorithm with respect to time complexity, memory requirement and data communication. Its cost is bounded by MTTKRP and its associated communications. For the sake of simplicity, we take a N-order tensor $\mathcal{X} \in \mathbb{R}^{I\times\cdots\times I}$ as the input tensor. We denote M as the number of machines, p as the number of partitions for one mode, $P = p\times p\times p$ as the number of blocks in a tensor and K as the number of components in the eigen-decomposition of a graph Laplacian matrix $\mathcal{L}$.

Lemma 1: The time complexity of DISTENC is $\mathcal{O}(nnz(\mathcal{X}) + NI + NIR + Rnnz(\mathcal{X}) + N(IR + IKR + IK^2R) + N(IR^2 + \lceil nnz(\mathcal{X})/I\rceil R + 3IR + R^3) + NIR)$.

Proof: In the beginning, the tensor $\mathcal{X}$ is split into P blocks by applying Algorithm 2. For each mode, computing the number of non-zero elements in slices takes $\mathcal{O}(nnz(\mathcal{X}))$ time via incremental computations that employ prior summation results. Identifying the partition boundaries for each mode takes $\mathcal{O}(I)$ time. Since a non-zero element is determined and mapped to a machine in a constant time based on identified boundaries, decentralizing all non-zero elements in $\mathcal{X}$ to blocks in machines takes $\mathcal{O}(nnz(\mathcal{X}))$. In total, partitioning the sparse tensor takes $\mathcal{O}(nnz(\mathcal{X})+NI)$. After splitting the tensor into blocks and mapping all non-zero elements to blocks, the factor matrices are randomly initialized and distributed among machines based upon block boundaries identified during the split of the tensor. This process takes $\mathcal{O}(NIR)$ time. The residual tensor $\mathcal{E}$ is sparse with the same number of non-zero elements as the input sparse tensor $\mathcal{X}$. Bounded by the non-negative weight tensor $\mathbf{\Omega}$, calculating the residual tensor takes $\mathcal{O}(Rnnz(\mathcal{X}))$ time as an entry of $[\![\mathbf{A}^{(1)},\ldots,\mathbf{A}^{(N)}]\!]$ can be obtained in $\mathcal{O}(R)$ time. Pre-computing the truncated eigen-decomposition of a graph Laplacian matrix $\mathcal{L}_n$ for n-mode takes $\mathcal{O}(KI)$ time. The error between a factor matrix $\mathbf{A}^{(n)}$ and a matrix of Lagrange multipliers $\mathbf{Y}^{(n)}$ can be computed in $\mathcal{O}(RI)$ time. Based upon the order of updating $\mathbf{B}^{(n)}$ introduced in the section III-B, computing the multiplication of last two matrices $\mathbf{V}_n^T(\eta_t\mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)})$ takes $\mathcal{O}(IKR)$. Due to the relatively small size ($K\times R$) of the result, we broadcast it with $(\eta_t + \alpha_n\Lambda_n)^{-1}$ of size $K\times K$ to the first matrix $\mathbf{V}_n$ and compute their multiplication in $\mathcal{O}(IK^2R)$ time. In total, updating an auxiliary variable takes $\mathcal{O}(IR + IKR + IK^2R)$ time, and $\mathcal{O}(N(IR + IKR + IK^2R))$ time for all modes. The update of a factor matrix contains three steps. The self-product $\mathbf{A}^{(n)T}\mathbf{A}^{(n)}$ for n-mode requires to be performed in $\mathcal{O}(IR^2)$ time. Through all N modes in the tensor, it takes $\mathcal{O}(NIR^2)$ time. In each mode, computing MTTKRP of the residual tensor and factor matrices as shown in line 7 of Algorithm 3 by the proposed row-wise method takes $\mathcal{O}(\lceil nnz(\mathcal{X})/I\rceil R)$. As illustrated in line 8 of Algorithm 3,

Algorithm 3: DisTenC Algorithm

```
   Input: $\mathcal{T}, \mathbf{A}_0^{(n)}, \mathbf{\Omega}, \mathbf{\Omega}^c, \lambda, \rho, \eta, \eta_{max}, N$
   Output: $\mathcal{X}, \mathbf{A}^{(n)}, \mathbf{B}^{(n)}, \mathbf{Y}^{(n)}$
1  for $n \leftarrow 1$ to $N$ do
2      $\boldsymbol{w}_n \leftarrow GreedyAlgorithm(\mathcal{X})$ in Algorithm 2
3  Partition $\mathcal{X}$ based upon $\boldsymbol{w}_n, n = 1,\cdots,N$
4  Initialize $\mathbf{A}_0^{(n)} \geq 0$, $\mathbf{B}_0^{(n)} = \mathbf{Y}_0^{(n)} = 0$, $t = 0$
5  Calculate the residual tensor
       $\mathcal{E}^0 = \mathbf{\Omega} * (\mathcal{T} - [\![\mathbf{A}_0^{(1)},\ldots,\mathbf{A}_0^{(N)}]\!])$
6  for $t \leftarrow 0$ to $T$ do
7      for $n \leftarrow 1$ to $N$ do
8          Update
               $\mathbf{B}_{t+1}^{(i)} \leftarrow (\eta_t\boldsymbol{I} + \alpha_n\mathcal{L}_n)^{-1}(\eta_t\mathbf{A}_t^{(n)} - \mathbf{Y}_t^{(n)})$
9          Calculate and cache $\mathcal{F}_n^t = \mathbf{U}_t^{(n)T}\mathbf{U}_t^{(n)}$
10         Calculate $\mathbf{H}_n^t = MTTKRP(\mathcal{E}_{(n)}^t\mathbf{U}_t^{(n)})$
11         Update and cache $\mathbf{A}_{t+1}^{(n)} \leftarrow (\mathbf{A}_t^{(n)}\mathcal{F}_n^t + \mathbf{H}_n^t +$
               $\eta_t\mathbf{B}_{t+1}^{(n)} + \mathbf{Y}_t^{(n)})(\mathcal{F}_n^t + \lambda\boldsymbol{I} + \eta_t\boldsymbol{I})^{-1}$
12         Update and cache
               $\mathbf{Y}_{t+1}^{(n)} = \mathbf{Y}_t^{(n)} + \eta_t(\mathbf{B}_{t+1}^{(n)} - \mathbf{A}_{t+1}^{(n)})$
13     Calculate and cache the residual tensor
           $\mathcal{E}^{t+1} = \mathbf{\Omega} * ([\![\mathbf{A}_{t+1}^{(1)},\ldots,\mathbf{A}_{t+1}^{(N)}]\!] - [\![\mathbf{A}_t^{(1)},\ldots,\mathbf{A}_t^{(N)}]\!])$
14     Update $\eta_{t+1} = \min(\rho\eta_t, \eta_{max})$
15     Check the convergence:
           $\max\{\|\mathbf{A}_{t+1}^{(n)} - \mathbf{A}_t^{(n)}\|_F^2\} < tol$
16     if converged then
17         break out of for loop
18 return $\mathcal{X}$, $\mathbf{A}^{(n)}, n = 1,2,\ldots,N$
```

updating a factor matrix performs a multiplication of two matrices. The first one can be obtained in $\mathcal{O}(IR^2+3IR)$ time. For the second one, it takes $\mathcal{O}(R^3)$ to calculate the inverse of the matrix $(\mathcal{F}_n^t + \lambda \boldsymbol{I} + \eta_t \boldsymbol{I})$. The multiplication of these two matrices takes $\mathcal{O}(IR^2)$. Thus, updating a factor matrix takes $\mathcal{O}(IR^2 + \lceil nnz(\mathcal{X})/I \rceil R + 3IR + R^3)$. In total, it takes $\mathcal{O}(N(IR^2 + \lceil nnz(\mathcal{X})/I \rceil R + 3IR + R^3))$. Updating the matrix of Lagrange Multiplier $\mathbf{Y}^{(n)}$ takes $\mathcal{O}(IR)$ time. Checking the convergence criterion requires $\mathcal{O}(NIR)$ to be performed. ■

Lemma 2: The amount of memory required by DisTenC is $\mathcal{O}(nnz(\mathcal{X}) + 3NIR + NIK + NK + MNR^2)$.

Proof: During the computation, DisTenC needs to store data in memory at each iteration as follows: the observed tensor $\mathcal{X}$, the residual tensor $\mathcal{E}$, factor matrices $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$, and $\mathbf{A}^{(3)}$, auxiliary variables $\mathbf{B}^{(1)}$, $\mathbf{B}^{(2)}$, and $\mathbf{B}^{(3)}$, Lagrange multiplier matrices $\mathbf{Y}^{(1)}$, $\mathbf{Y}^{(2)}$, and $\mathbf{Y}^{(3)}$, eigen-decomposed graph Laplacian matrix $\mathcal{L}_n = \mathbf{V}_n \mathbf{\Lambda}_n \mathbf{V}_n^T$, and the self-product ${\mathbf{A}^{(n)}}^T \mathbf{A}^{(n)}$ for n-mode. Since the residual tensor $\mathcal{E}$ is calculated only for those non-zero elements in the observed tensor $\mathcal{X}$, both of them are kept in the memory at each iteration with a distributed fashion, which require $\mathcal{O}(nnz(\mathcal{X}))$ memory. For each mode, its factor matrix $\mathbf{A}^{(n)}$ has the same size as its auxiliary variable $\mathbf{B}^{(n)}$ and Lagrange multiplier matrix $\mathbf{Y}^{(n)}$. Thus, the total amount of memory used for storing them for all modes is $\mathcal{O}(3NIR)$. The Laplacian matrix $\mathcal{L}_n$ for n-mode is eigen-decomposed into an eigenvector matrix $\mathbf{V}_n$ and a diagonal matrix $\mathbf{\Lambda}_n$ that is stored as a vector in the machine. By considering all modes in a tensor, the memory is required to hold $\mathcal{O}(NIK + NK)$ space. The self-product ${\mathbf{A}^{(n)}}^T \mathbf{A}^{(n)}$ for n-mode only takes $\mathcal{O}(R^2)$ memory. Since we will broadcast it to all M machines, the amount of memory for storing these self-products for all modes is $\mathcal{O}(MNR^2)$. ■

Lemma 3: The amount of shuffled data caused by DisTenC is $\mathcal{O}(nnz(\mathcal{X}) + TNMIR + TNMR^2)$

Proof: DisTenC initially employed the greedy algorithm to identify the partition boundaries for each mode, and partitions the observed tensor $\mathcal{X}$ into defined groups. In this process, the whole input tensor is shuffled and cached across all machines. Thus, the amount of shuffled data in the partition of the observed tensor is $\mathcal{O}(nnz(\mathcal{X}))$. In each iteration, DisTenC requires to send rows of factor matrices, auxiliary variables and Lagrange multiplier matrices to corresponding partitions in which each tensor entry is updated by rows associated with its index in all modes, which takes $\mathcal{O}(NMIR)$ space in total. Moreover, the self-product ${\mathbf{A}^{(n)}}^T \mathbf{A}^{(n)}$ for n-mode is calculated by aggregating all matrices of size $R \times R$ across machines, and then broadcast back to all machines. In this process, it takes $\mathcal{O}(NMR^2)$ by considering all modes in the tensor. Similarly, updating the residual tensor $\mathcal{E}$ needs to copy associated rows of factor matrices into machines, which takes $\mathcal{O}(NMIR)$ space in sum. Therefore, by considering all cases above, the amount of shuffled data for DisTenC after T iterations is $\mathcal{O}(nnz(\mathcal{X}) + TNMIR + TNMR^2)$. ■

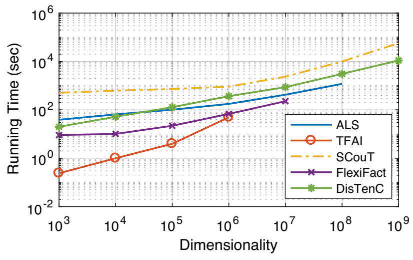
F. Implementation on Spark

In this section, we explore practical issues in terms of implementations of DisTenC on Spark. Our implementation is carefully designed to obtain best speed-up and scalability. Since the input tensor is sparse, all entries are stored in a list with the coordinate format (COO). The input sparse tensor is loaded as RDDs. First of all, we apply functions `map` and `reduceByKey` to calculate the number of non-zero elements for all indices in a mode with the key that is the index in that mode. These count results are then used to generate partition boundaries for that mode and `persist`ed in memory. After that, we apply functions `map` and `aggregateByKey` to partition the tensor into blocks: `map` transforms an entry of the sparse tensor into an element in the RDD whose key is a block ID; `aggregateByKey` groups these non-zero elements by block IDs. Partitioned tensor RDDs are then `persist`ed in memory. In order to speed-up the following computation, for each mode, we transform a tensor to a pair RDD whose key is an index in that mode and value is all block IDs with which entries associated with this index appear in blocks by employing RDD's functions `flatMap` and `reduceByKey`, and `persist` them in memory. Factor matrices are initialized with random numbers, which are stored as RDDs and distributed based upon partition boundaries identified previously. Following the same fashion, matrices of Lagrange multipliers are initialized with zeros as RDDs. After applying the efficient truncated eigen-decomposition method, a graph Laplacian matrix is decomposed into eigenvalues and eigenvectors. As shown in the section III-B, a diagonal matrix of eigenvalues is stored as a `Array` and broadcast to all machines in the cluster; eigenvectors are stored as RDDs where the key is the index and the value is the associated eigenvector with the same partition as factor matrices. The self-product of a factor matrix is transformed from a factor matrix RDD by utilizing functions `flatMap` and `reduceByKey` and broadcast to all machines. We update factor matrices as well as auxiliary variables by using RDD's functions `flatMap`, `join` and `reductByKey`. Since the operation `join` will shuffle the data and exponentially increase the computational time, we keep the same partitions when applying `join` to two RDDs. In the implementation, we also replace operations `groupByKey` by `reduceByKey` and `combineByKey` that combines pairs with the same key on the same machine for efficiency. We also limit the number of `combineByKey` operations so that edges of the same element are available at the same physical location, minimizing data shuffling. As it can be seen, we cache reused RDD in memory in order to minimize disk accesses between consecutive iterations, which would not be possible if using a system like Hadoop to distribute the computation.

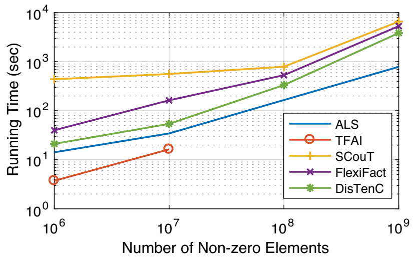
IV. Experiments

To evaluate the proposed DisTenC, we perform experiments to answer the following questions:

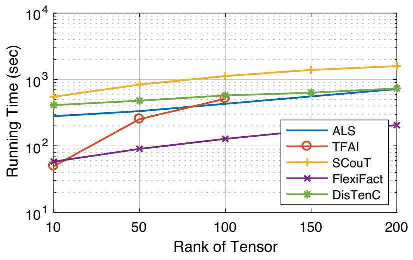
Q1: Data Scalability. How well do DisTenC and other baseline methods scale up with the input tensor in terms of



(a) Dimensionality



(b) Number of Non-zeros



(c) Rank

Fig. 3: Data Scalability of our proposed DISTENC compared with other methods. DISTENC successfully complete tensors with high dimensionality, sparsity and rank, while the baseline methods fail running out of time or memory. Concretely, DISTENC is capable of handling $10 \sim 1000\times$ larger tensors, addressing $100\times$ denser tensors and less effects on rank.

factors such as the number of non-zeros, dimension, mode length, and rank?

Q2: Machine Scalability. How well does DISTENC scale up in terms of the number of machines?

Q3: Discovery. How accurately do DISTENC and other baseline methods perform over real-world tensor data?

A. Experimental Setup

Cluster/Machines. DISTENC is implemented on a 10-node Spark cluster in which each node has a quad-core Intel Xeon E5410 2.33GHz CPU, 16GB RAM and 4 Terabytes disk. The cluster runs Spark v2.0.0 and consists of one driver node and 9 worker nodes. In the experiments, we employ 9 executors, each of which uses 8 cores. The amount of memory for the driver and each executor process is set to 8GB and 12GB.

Datasets. Both synthetic and real-world data are used to evaluate the proposed method. We generate two synthetic datasets, one for testing the scalability and the other for testing the reconstruction error. For the scalability tests, we generate random tensors of size $I \times J \times K$ by randomly setting a data point at (i, j, k). For simplicity, we assume that their similarity matrices are identity matrices for all modes. For the reconstruction error tests, we first randomly generate factor matrices $\mathbf{A}^{(1)}$, $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ with the specific rank $R = 20$ by the following linear formula [14]:

$$\mathbf{A}^{(1)}_{i,r} = i\varepsilon_r + \varepsilon', \qquad i = 1, 2, \ldots, I_1, r = 1, 2, \cdots, R$$
$$\mathbf{A}^{(2)}_{j,r} = j\zeta_r + \zeta', \qquad j = 1, 2, \ldots, I_2, r = 1, 2, \cdots, R$$
$$\mathbf{A}^{(3)}_{k,r} = k\eta_r + \eta', \qquad k = 1, 2, \ldots, I_3, r = 1, 2, \cdots, R$$

where $\{\varepsilon_r, \varepsilon'_r, \zeta_r, \zeta'_r, \eta_r, \eta'_r\}_{r=1,2,\ldots,R}$ are constants generated by the standard Gaussian distribution $N(0, 1)$. Since each factor matrix is generated by linear functions mentioned above column by column, the consecutive rows are similar to each other. Therefore, we generate the similar matrix for the ith mode as the following tri-diagonal matrix:

$$\mathbf{S}_i = \begin{bmatrix} 0 & 1 & 0 & \ldots \\ 1 & 0 & 1 & \ldots \\ 0 & 1 & 0 & \ldots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix} \qquad (17)$$

We then randomly select tensor data points (i, j, k) as our observation and calculate its value via $\mathbf{A}^{(1)}_{i,:} \circ \mathbf{A}^{(2)}_{j,:} \circ \mathbf{A}^{(3)}_{k,:}$. This process is performed until we have the desired number of observed data points. We vary the dimensionality of the synthetic data as well as the rank in order to test the scalability and the reconstruction error, respectively.

For real-world datasets, we use Netflix, Facebook, DBLP, and Twitter summarized in Table II with the following details:

- **Netflix:** Movie rating data employed in the Netflix prize [19], forming a *user-movie-time* tensor data by considering the time at which a user rated a movie. The rating ranges from 1 to 5.
- **Facebook:** Temporal relationships between users from the Facebook New Orleans networks [20], where We consider a 3-order tensor where the third mode corresponds to the date when one anonymized user adds the other user to the first user's friend list.
- **DBLP:** A record of DBLP (a computer science bibliography) publications including authors, papers, and conferences. We convert the dataset into a a co-authorship network with *author-author-paper* elements, and define a *author-author* similarity based on whether they come from the same affiliation.
- **Twitter:** Geo-tagged Twitter lists data [21]. A Twitter list allows a user (creator) to label another user (expert) with an annotation (e.g., news, food, technology). Since there are a large number of annotations, we transfer them into 16 general topics like news, music, technology, sports, etc. We convert relationships between list creators and experts into a 3-dimensional tensor by adding the topics of lists as the third mode, and produce a *creator-expert* similarity matrix based on their following relationships.

Baseline Methods. We compare DISTENC with two tensor completion methods and two state-of-the-art distributed matrix-tensor factorization methods. For the tensor completion methods, we consider; (i) ALS [22], a distributed tensor completion method based upon the alternating least square (ALS) with MPI and OpenMP; and (ii) TFAI [14], a single-machine tensor completion method with the integration of auxiliary information. We use the original implementation of ALS. Since the other completion method CCD++ based upon

TABLE II: Summary of the real-world and synthetic datasets used. **K**: thousand, **M**: million, **B**: billion.

Datasets	I	J	K	Non-zeros
Netflix	480K	18K	2K	100M
Facebook	60K	60K	5	1.55M
DBLP	317K	317K	629K	1.04M
Twitter	640K	640K	16	1.13M
Synthetic-scalability	1K∼1B	1K∼1B	1K∼1B	10K ∼10B
Synthetic-error	10K	10K	10K	10M

the circle coordinate descent [22] has a similar performance with ALS, we only consider ALS as one of our baseline methods in this paper. For the two distributed matrix-tensor factorization methods, we consider: (i) SCouT [23]; and (ii) FlexiFact [10] implemented on MapReduce. We integrate the similarity matrices of all modes as coupled matrices into SCouT and FlexiFact, respectively, as the way we adopted here.

B. Data Scalability

We employ synthetic tensors to evaluate the scalability of DisTenC comparing with other baseline methods in terms of three aspects: dimensionality, the number of non-zeros and rank. For the sake of simplification, we set the similarity matrices of all modes to the identity matrices in the scalability tests. All experiments are allowed to run 8 hours. If methods cannot conduct the results within 8 hours, they will be marked as Out-Of-Time (OOT).

Dimensionality. We increase the tensor size $I = J = K$ from 10^3 to 10^9 while fixing the rank to 20 and the number of non-zero elements to 10^7. As shown in Fig. 3a, DisTenC and SCouT outperform other baseline methods by successfully performing tensor completion on tensors of size $I = J = K = 10^9$. On the other hand, both ALS and FlexiFact run with the out-of-memory (O.O.M.) error when $I = J = K \geq 10^7$; TFAI causes the O.O.M. error when $I = J = K \geq 10^6$. FlexiFact does not scale up for very large datasets due to its high communication cost with an exponential increase. ALS requires each communication of entire factor matrices per epoch in the worst case as a coarse-grained decomposition. TFAI is bounded by the memory of a single machine.

Number of Non-Zeros. We increase the number of non-zero elements (density) from 10^6 to 10^9 while fixing the dimensionality of the input tensor to $I = J = K = 10^5$ and the rank to 10. As demonstrated in Fig. 3b, only TFAI runs out of memory due to the bound of a single machine while other methods including the proposed DisTenC, ALS, SCouT and FlexiFact are able to scale up to 10^9 non-zero elements. DisTenC takes more running time than ALS with shrinked differences as the number of non-zero elements increases. But DisTenC outperforms both SCouT and FlexiFact due to the advantages of Spark that is more fit for running the iterative algorithms with less disk accesses.

Rank. We increase the rank of a tensor from 10 to 500 while fixing the dimensionality to $I = J = K = 10^6$ and the

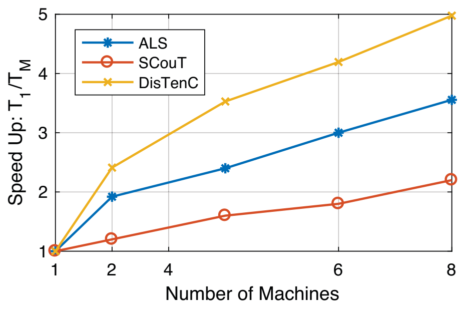


Fig. 4: Machine scalability of DisTenC compared with ALS and SCouT. The proposed DisTenC has the best performance in terms of machine scalability with $4.9\times$ speed-up, which also achieves a better linearity on the scalability with respect to the number of machines.

number of non-zero elements to 10^7. As shown in Fig. 3c, all methods except of TFAI are capable of scaling up to rank 200. The running time of ALS rapidly increases when the rank becomes large due to its cubically increasing computational cost. DisTenC has a relatively flat curve as the increase of rank due to its optimization on calculating the inverse of a symmetric matrix.

C. Machine Scalability

We measure the machine scalability of the proposed DisTenC by increasing the number of machines from 1 to 8. The synthetic dataset of size $I = J = K = 10^5$ with 10^7 non-zero elements is applied and its rank is set to 10. In Fig. 4, we report the ratio T_1/T_M where T_M is the running time using M machines. Since TFAI is a single-machine tensor completion method and FlexiFact has a worse scalability on machines than SCouT [23], we only compare ALS, SCouT and the proposed DisTenC. It can be seen that DisTenC obtains $4.9x$ speed-up as increasing the number of machines from 1 to 8 and achieves a better linearity in terms of machine scalability, which SCouT slows down due to the intensive hard disk accesses and high communication cost.

D. Reconstruction Error

In order to evaluate the accuracy of the proposed DisTenC with respect to the reconstruction error, we use the synthetic dataset of size $I = J = K = 10^4$ with 10^7 non-zero elements and set its rank to 10, and adopt adopt *Relative Error* as our evaluation metric. Relative Error is defined as $RelativeError = \|X - Y\|_F / \|Y\|_F$, where X is the recovered tensor and Y is the ground-truth tensor. We randomly sample the non-zero elements based upon the missing rate as the testing data to calculate the relative error; the rest is used as the training data. We report results in Fig. 5 by varying the missing rate from 30%, 50% and 70%. All results are averaged by running 5 times in order to reduce the dependency of randomness. Overall, we witness that DisTenC achieves comparable performance with TFAI, but better than ALS, and SCouT though the relative errors for all methods are relatively high due to the extreme sparsity of the data. We see that the

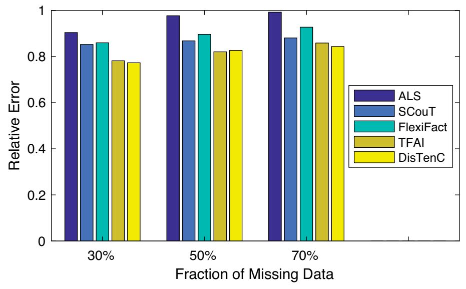


Fig. 5: Reconstruction error on the synthetic data.

integrated auxiliary information (similarity matrix) leads to significant improvement through the tensor completion. These relationships can alleviate the problem of sparsity to some extent and provide valuable information for the tensor completion to obtain more interpretable low-rank representations.

E. Recommender System

In this section, we apply DISTENC to perform recommendation on large scale real-world datasets and present our findings. We are mostly interested in illustrating the power of our approach rather than systemically comparing with all other state-of-the-art methods. Since SCOUT has a better scalability than FLEXIFACT and TFAI cannot handle such large-scale datasets, we only compare out proposed DISTENC with other two baseline methods ALS and SCOUT. The root-mean-square error (RMSE) is adopted as our evaluation metric, which represents the sample standard deviation of the differences between observed tensor $\mathfrak{T}$ and predicted tensor $\mathfrak{X}$ as $RMSE = \sqrt{\|\mathbf{\Omega} * (\mathfrak{T} - \mathfrak{X})\|_F^2 / nnz(\mathfrak{T})}$. It has been commonly used in the evaluation of recommender systems. We randomly use 50% of the observation for training, and the rest for testing. All results are reported by running 5 times and computing the average performance.

Netflix. We conduct the recommendation on Netflix dataset which contains a *user-movie-time* tensor and a *movie-movie* similarity matrix generated based on the movie title. As shown in Fig. 6a, we observe that the proposed DISTENC obtains the best performance in the precision of recommendation an average improvement of 14.9% over other baseline methods. In addition, by introducing the auxiliary information, both DISTENC and SCOUT outperform ALS. On the other hand, Fig. 6b shows that the proposed DISTENC converges the fastest to the best solution by taking advantage of ADMM [15], [16]. SCOUT takes much longer time on the convergence by employing the MAPREDUCE framework.

Twitter. Using DISTENC, we also perform the expert recommendation on Twitter List dataset which consists of a *creator-expert-topic* tensor as well as *creator-creator* and *expert-expert* similarity matrices calculated based on whether they are located in the same location (cities). As demonstrated in Fig. 6a, DISTENC performs the best among all alternative baseline methods with an average improvement of 21.4% in the precision. Concretely, DISTENC outperforms ALS with an improvement of 32.6% in precision, indicating the the superiority of a tensor completion model integrating auxiliary information. With respect to the convergence, DISTENC has similar performance as one on the Netflix dataset shown in Fig. 6b. Due to the limited space, we omit its details.

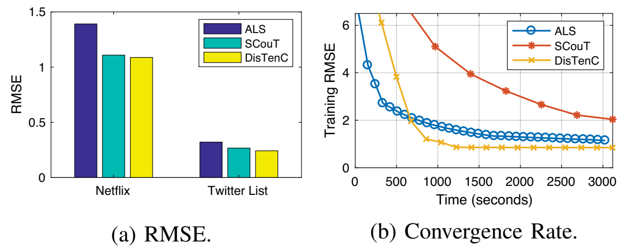


Fig. 6: Results on recommender system: (a) RMSE on Netflix and Twitter List datasets; (b) Convergence rate for all methods on the Netflix data.

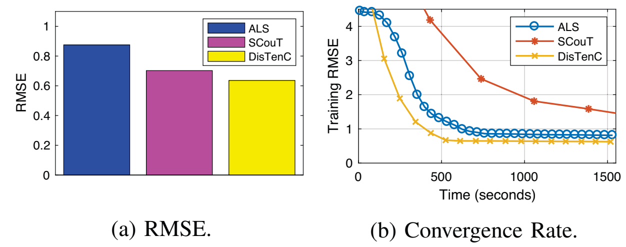


Fig. 7: Results on link prediction: (a) RMSE on the Facebook dataset; (b) Convergence rate for all methods.

F. Link Prediction

As one of the most applications for tensor completion, link prediction aims to recover unobserved links between nodes in a low-rank tensor (the matrix is a special case). Using DISTENC, we perform link prediction on Facebook dataset that includes a *user-user-time* tensor and a similarity matrix *user-user* generated based on the similarities between their wall posts. As a similar fashion in the previous section, we randomly select 50% of observations for training, and the rest for testing. We also adopt RMSE as the evaluation metric in this experiment. To reduce statistical variability, experimental results are averaged by running 5 times. Fig. 7 illustrates the testing accuracy and the training convergence. As we can see, both DISTENC and SCOUT have comparable performance and are better than ALS in precision. Specifically, DISTENC outperforms ALS with an average improvement of 27.4%; SCOUT has a better performance than ALS with an average improvement of 19.5%. In terms of convergence, DISTENC converges faster to the best solution.

G. Discovery

Since tensor completion performs both imputation and factorization meanwhile, we apply DISTENC on DBLP dataset that contains a *author-paper-venue* tensor with a similarity matrix *author-author*. We randomly select 50% of observations for training the model. After that, we pick top-k highest valued elements from each factor after filtering too general elements. We show 3 notable concepts we found in Table III. It can be seen that all conferences within a concept are correlated and all famous researchers in each concept are discovered.

TABLE III: Example of concept discovery results on DBLP.

Concept	Authors	Conferences
Database	Surajit Chaudhuri, Michael J. Carey, David J. DeWitt, Rajeev Rastogi, Dan Suciu, Ming-Syan Chen	SIGMOD VLDB ICDE
Data Mining	Jiawei Han, Philip S. Yu, George Karypis, Christos Faloutsos, Shusaku Tsumoto, Rakesh Agrawal	KDD ICDM PKDD
Info. Retrieval	W. Bruce Croft, Mark Sanderson, Iadh Ounis, ChengXiang Zhai, Gerard Salton, Clement T. Yu	SIGIR ECIR WWW

V. Related Work

In this section, we review related works on tensor completion, scalable tensor algorithms, and distributed computing frameworks.

A. Tensor Completion

Tensor factorization models have been studied and applied in many fields due to their strong power on multi-dimensional data analysis. There are two widely used low-rank decompositions of tensors, the CANDECOMP/PARAFAC (CP) and the Tucker decompositions [24]. The most common methods used to factorize tensors include alternating least square (ALS) [25], [26], [27], stochastic gradient descent (SGD) [10], [11] and coordinate descent (CDD) [28], [9]. Tensor completion is used to estimate missing values in tensors based on their low-rank approximations, which has been extensively studied and employed in applications such as recommendations [21], [1], user group detection [13], and link prediction [29]. Most these methods only focus on the sampled data when performing tensor completion without considering auxiliary information with which tensors usually come. These auxiliary information help us have a better performance in tensor completion [30]. Though several researchers incorporate auxiliary information into the matrix factorization problem [31], few studies explore the tensor completion problem with auxiliary information. Technically, it is challenging to embed auxiliary information into a completion model, especially with many heterogeneous contexts. Narita et al. [30] integrated auxiliary information into tensor decomposition methods, resulting in better performance compared with ordinary tensor decomposition methods. Nevertheless, they primarily focus on general tensor decomposition rather than tensor completion. However, these models usually face some efficiency challenges since [30] requires solving the Sylvester equation with a high cost several times in each of iterations, making them infeasible for large-scale applications.

B. Scalable Tensor Factorization

We witness considerable efforts on developing scalable algorithms for tensor factorization, most of which focus on solving the intermediate data explosion problem. Concretely, pioneers Bader and Kolda [26] develop efficient algorithms for sparse tensor decomposition by avoiding the materialization of very large, unnecessary intermediate Khatri-Rao products. Kolda and Sun [6] continuously work on the specific tensor decomposition method Tucker for sparse data and solve the intermediate explosion problem by calculating the tensor-matrix multiplication one slice or fiber at a time. An alternative approach, DBN, is introduced in [32] where the authors use Relational Algebra to break down the tensor into smaller tensors, using relational decomposition, and thus achieving scalability. Kang et al. [7] first propose a scalable distributed algorithm GigaTensor under the MapReduce framework for the specific tensor decomposition method PARAFAC by decoupling the Khatri-Rao product and calculate it distributively column by column. Jeon et al. [33] improves on GigaTensor and propose HaTen2 that is a general, unified framework for both Tucker and CP tensor decomposition. There has been other alternatives on solving the intermediate data explosion problem of the pure tensor composition [34], [35], [36].

On the other hand, some researchers put their focus on developing scalable, distributed algorithms for the tensor decomposition with additional side information that is usually represented in a matrix, e.g., a similarity matrix between experts. Papalexakis et al. [11] propose an efficient scalable framework to solve the coupled matrix-tensor factorization problem by leveraging the biased sampling to split the original large data into samples, running the common solver to samples and merging the results based on the common parts in each sample. Beutel et al. [10] propose FlexiFact, a MapReduce algorithm to decompose matrix, tensor, and coupled matrix-tensor based on stochastic gradient descent. Jeon et al. [23] propose SCouT for scalable coupled matrix-tensor factorization. Shin et al. [9] propose two scalable tensor factorization algorithms SALS and CDTF based on subset alternating least square and coordinate descent, respectively. Livas et al. [16] develop a constrained tensor factorization framework based on ADMM. Smith et al. [22] optimize and evaluate three distributed tensor factorization algorithms based on ALS, SGD and CDD, respectively, by extending SPLATT [8] that optimizes the memory usage. Despite the extensive research efforts that have been devoted to tensor factorization, as reviewed above, distributed tensor completion using auxiliary information has yet received much attention, especially on modern distributed computing frameworks such as Spark. Therefore we believe our work fills an important gap in tensor-based mining algorithms.

C. Distributed Computing Frameworks

MapReduce [37] is a distributed computing model for processing large-scale datasets that cannot be handled in a single machine, running in a massively parallel manner. MapReduce has been the most popular distributed computing framework due to its advantages including automatic data distribution, fault tolerance, replication, massive scalability, and functional programming by which users only define two functions `map` and `reduce`. Hadoop [38] is an open-source of MapReduce. Due to its excellent scalability and ease of use, it has been successfully applied in many data mining applications [7], [39], [40]. However, Hadoop is inefficient to execute iterative algorithms due to its intensive disk accesses [41]. Apache Spark [42] is an in-memory MapReduce, providing a high-level interface for users to build applications with respect to large-scale data computation. Spark allows to store intermediate data in memory and performs efficient memory-based operations without requiring data to be spilled to disk (effectively reducing the number of disk Input/Output operations). Therefore, Spark is capable of performing iterative algorithms very efficiently. Due to these advantages, Spark has been used in applications [35].

VI. Conclusion

In this paper, we propose DisTenC, a distributed algorithm for tensor completion with the integration of auxiliary information based on ADMM, which is capable of scaling up to billion size tensors and achieving good performance across many applications. By efficiently handling trace-based regularization term, updating factor matrices with caching, and optimizing the update of new tensor, DisTenC successfully addresses the high computational cost, and minimizes the generation and shuffling of the intermediate data. Through extensive experiments, DisTenC shows up to $100x$ larger scalability than existing methods, and converges much faster than state-of-the-art methods. In addition, DisTenC obtains an average improvement of 18.2% on a recommender system scenario and 23.5% on link prediction.

Acknowledgements. Materials in this paper have been supported by the DARPA (W911NF-16-1-0565).

References

[1] S. Rendle and L. Schmidt-Thieme, "Pairwise interaction tensor factorization for personalized tag recommendation," *WSDM*, 2010.
[2] R. Yu and Y. Liu, "Learning from multiway data: Simple and efficient tensor regression," in *ICML*, 2016, pp. 238–247.
[3] Z. Zhao, Z. Cheng, L. Hong, and E. H. Chi, "Improving user topic interest profiles by behavior factorization," in *WWW*, 2015.
[4] H. Ge, J. Caverlee, N. Zhang, and A. Squicciarini, "Uncovering the spatio-temporal dynamics of memes in the presence of incomplete information," in *CIKM*, 2016.
[5] F. Morstatter, J. Pfeffer, H. Liu, and K. M. Carley, "Is the sample good enough? comparing data from twitter's streaming api with twitter's firehose." in *ICWSM*, 2013.
[6] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *IEEE ICDM*, 2008.
[7] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries," *SIGKDD*, 2012.
[8] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *IPDPS*, 2015.
[9] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," *ICDM*, 2014.
[10] 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.
[11] 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.
[12] F. L. Hitchcock, "The expression of a tensor or a polyadic as a sum of products," *Journal of Mathematics and Physics*, 1927.
[13] Y. Liu, F. Shang, L. Jiao, J. Cheng, and H. Cheng, "Trace norm regularized candecomp/parafac decomposition with missing data," *IEEE transactions on cybernetics*, 2015.
[14] A. Narita, K. Hayashi, R. Tomioka, and H. Kashima, "Tensor factorization using auxiliary information," *Data Mining and Knowledge Discovery*, 2012.
[15] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine Learning*, 2011.
[16] A. P. Liavas and N. D. Sidiropoulos, "Parallel algorithms for constrained tensor factorization via alternating direction method of multipliers," *IEEE Transactions on Signal Processing*, 2015.
[17] P. Bientinesi, I. S. Dhillon, and R. A. Van De Geijn, "A parallel eigensolver for dense symmetric matrices based on multiple relatively robust representations," *SIAM Journal on Scientific Computing*, 2005.
[18] S. Smith and G. Karypis, "A medium-grained algorithm for distributed sparse tensor factorization," in *IPDPS*, 2016.
[19] J. Bennett, S. Lanning *et al.*, "The netflix prize," in *Proceedings of KDD cup and workshop*, 2007.
[20] B. Viswanath, A. Mislove, M. Cha, and K. P. Gummadi, "On the evolution of user interaction in facebook," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Social Networks (WOSN'09)*, 2009.
[21] H. Ge, J. Caverlee, and H. Lu, "Taper: a contextual tensor-based approach for personalized expert recommendation," *Proc. of RecSys*, 2016.
[22] S. Smith, J. Park, and G. Karypis, "An exploration of optimization algorithms for high performance tensor completion," in *SC for High Performance Computing, Networking, Storage and Analysis*, 2016.
[23] B. Jeon, I. Jeon, L. Sael, and U. Kang, "Scout: Scalable coupled matrix-tensor factorization-algorithm and discoveries," in *ICDE*, 2016.
[24] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM review*, 2009.
[25] J. H. Choi and S. Vishwanathan, "Dfacto: Distributed factorization of tensors," in *NIPS*, 2014, pp. 1296–1304.
[26] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, 2007.
[27] G. Tomasi and R. Bro, "Parafac and missing values," *Chemometrics and Intelligent Laboratory Systems*, 2005.
[28] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the cp format," *Parallel Computing*, 2016.
[29] D. M. Dunlavy, T. G. Kolda, and E. Acar, "Temporal link prediction using matrix and tensor factorizations," *TKDD*, 2011.
[30] A. Narita, K. Hayashi, R. Tomioka, and H. Kashima, "Tensor factorization using auxiliary information," *ECML PKDD*, 2011.
[31] R. Forsati, M. Mahdavi, M. Shamsfard, and M. Sarwat, "Matrix factorization with explicit trust and distrust side information for improved social recommendation," *TOIS*, 2014.
[32] M. Kim and K. S. Candan, "Decomposition-by-normalization (dbn): leveraging approximate functional dependencies for efficient tensor decomposition," in *CIKM*, 2012.
[33] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," *IEEE ICDE*, 2015.
[34] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the cp format," *Parallel Computing*, 2016.
[35] N. Park, S. Oh, and U. Kang, "Fast and scalable distributed boolean tensor factorization," in *ICDE*, 2017.
[36] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC for High Performance Computing, Networking, Storage and Analysis*, 2015.
[37] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, 2008.
[38] S. Wadkar, M. Siddalingaiah, and J. Venner, *Pro Apache Hadoop*. Apress, 2014.
[39] S. Papadimitriou and J. Sun, "Disco: Distributed co-clustering with map-reduce: A case study towards petabyte-scale end-to-end mining," in *ICDM*, 2008.
[40] G. D. F. Morales and A. Bifet, "Samoa: scalable advanced massive online analysis." *Journal of Machine Learning Research*, 2015.
[41] V. Kalavri and V. Vlassov, "Mapreduce: Limitations, optimizations and open issues," in *IEEE TrustCom*, 2013.
[42] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *USENIX on Networked Systems Design and Implementation*, 2012.